Old Dominion University Research Foundation

Design and Analysis of Architectures for Structural Health Monitoring Systems

Final Report

NASA Grant: NAG-1-2279

Principal Investigator:
Ravi Mukkamala
Department of Computer Science
Old Dominion University
Norfolk, Virginia 23529-0162

Design and Analysis of Architectures for Structural Health Monitoring Systems

Final Report

NASA Grant: NAG-1-2279

Principal Investigator:
Ravi Mukkamala
Department of Computer Science
Old Dominion University
Norfolk, Virginia 23529-0162

Summary of work done:

During the two-year project period, we have worked on several aspects of Health Usage and Monitoring Systems for structural health monitoring. In particular, we have made contributions in the following areas.

1. **Reference HUMS architecture:** We developed a high-level architecture for health monitoring and usage systems (HUMS). The proposed reference architecture is shown in Figure 1 [2]. It is compatible with the Generic Open Architecture (GOA) proposed as a standard for avionics systems [3]. It consists of six layers that may be logically divided into three parts. The lower part deals with sensors and low-level processing and control. The middle part deals with system level processing and maintenance. Finally, the upper part is related to the application software and interfacing with the user.

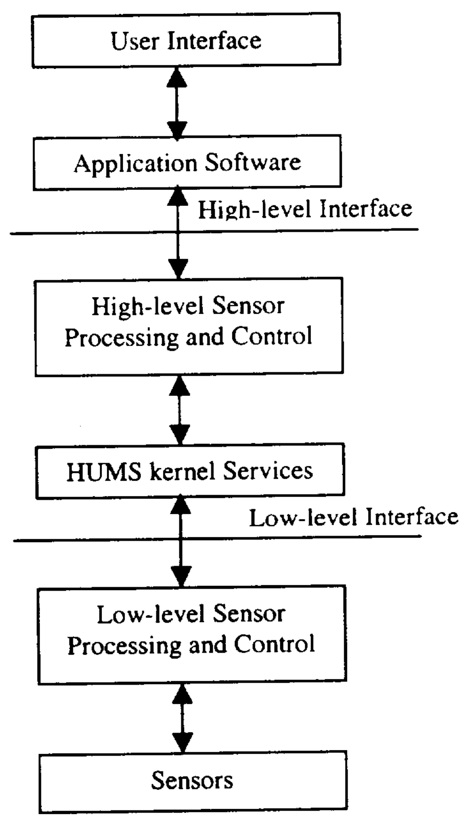


Figure 1. HUMS Reference Architecture

2. **HUMS kernel:** One of the critical layers of HUMS reference architecture is the HUMS kernel. We developed a detailed design of a kernel to implement the high level architecture [2,3]. The kernel provides the software infrastructure needed to

build HUMS architectures. It is designed with the objective of supporting scalability, robustness, and flexibility in HUMS. It offers the functionality needed to build a dynamic and robust distributed sensor system that is not usually offered by a COTS operating system. In particular, it offers services to support the following features.

- Dynamically add/delete/replace components such as sensors, services, processes, processors, and other system resources.
- Monitor resources and processes during their operation to detect any failures. It also has provisions to recover from process failures.
- Offer a hierarchical (i.e., logical) view of the system. This enables an application or an end-user to be transparent to low-level details (such as sensor Ids or number of sensors at a grid) and simply refer to high-level components (e.g., data values from all grids covering left-wing).
- Avoid strict synchronization between data producers (sensors) and data consumers (e.g., application/system processes). This enables to build more flexible systems.

A block diagram of the HUMS kernel that we have designed and implemented is shown in Figure 2. The kernel software is organized into five basic modules: lifecycle services, naming services, relationship services, buffering services, and monitoring & relocation services.

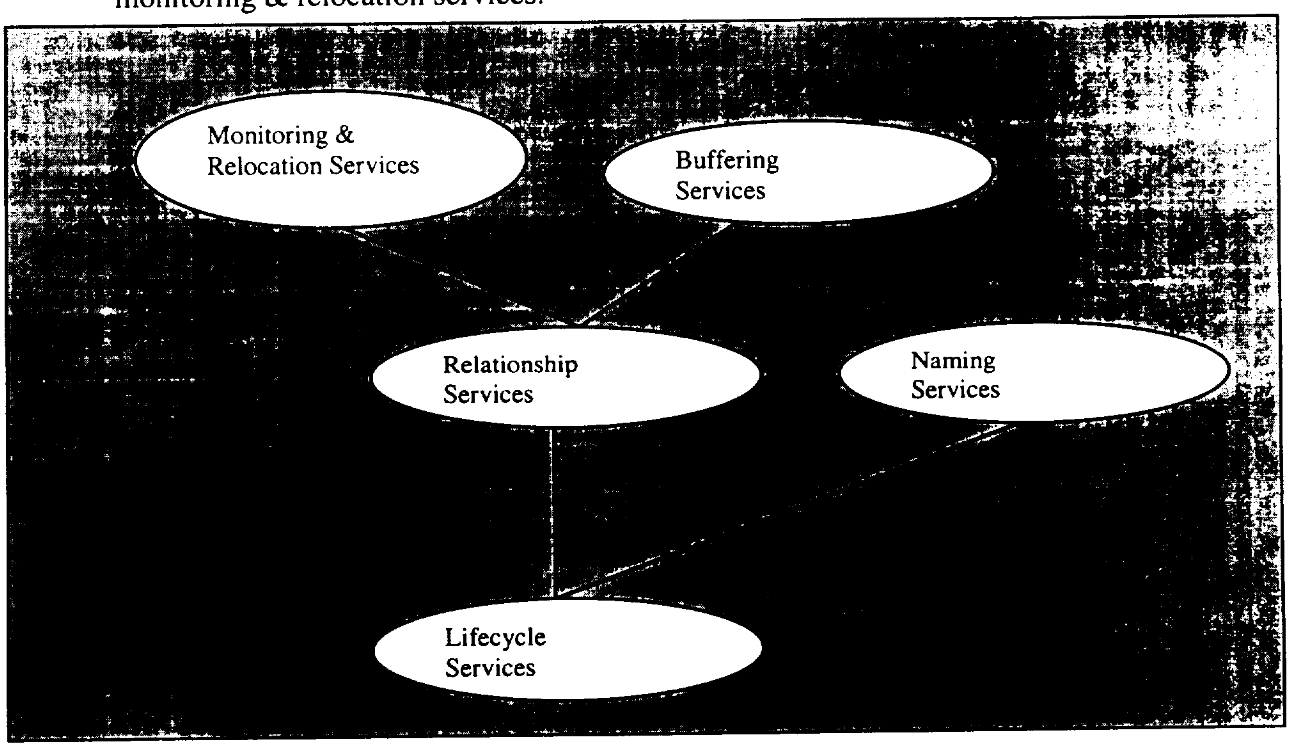


Figure 2. HUMS Kernel: A block diagram

3. **Prototype implementation of the HUMS kernel:** We have implemented a preliminary version of he HUMS kernel on a Unix platform. We have implemented both a centralized system version and a distributed version. In the centralized version, there is a single processor to which all sensors are connected. The sensors are simulated by means of data generating processes.

   In the distributed version, several processors in the system are connected using Ethernets. TCP/IP is used as a means of inter-process communication.
   In both cases, the relationship service is implemented using an Oracle database. Alternately, it could have been implemented as a simple file service also. For the administrator to interact with the lifecycle services, to add/delete services/processes/sensors, a Java-based GUI has been implemented.

   The sensors have been simulated by processes that emit signals at random (0-20sec) intervals to the sensor-controller. The communication between the sensor and the sensor-controller is implemented using UDP [4]. UDP was the chosen protocol because it uses *best-effort* delivery and does not impose the overhead of handshaking. This is similar to the real environment, where sensors produce signals irrespective of whether the communication medium is able to accommodate them or not. Additionally the sensors are not concerned about the signals reaching the destination.

   The sensor-controllers (SC) receive the signals from the sensors over a 20 second period. Each SC performs a simple sensor fusion (averaging) of the signals from each grid and transmits the signals to the buffering service that is located at a well-known address. The SCs transmit the signals to the buffering service in intervals of 20 seconds. The buffering service is located at well-known addresses (i.e., the ip and port of the buffering service is known to the sensor-controllers).

   The buffering service is duplicated. A primary buffering service and a secondary buffering service run at different machines and both their addresses are known to all the sensor-controllers. The communication between the sensor-controllers and the buffering service is achieved using TCP [4]. A failure of the sensor-controller to send signals to the buffering service would be interpreted as the failure of the buffering service. When such a failure occurs, the sensor-controller sends signals to another buffering service. The secondary buffering service is configured to monitor the primary. When the primary is alive the secondary remains dormant, but when the primary dies, the secondary becomes active. The primary has the highest priority to handle signals, hence if the primary is restarted again, it becomes active and the secondary shifts to the passive mode. Thus, the communication protocol between the sensor-controllers and the buffering service is provided with sufficient intelligence to determine failure of transmission of the signals.

   Currently, we are discovering means to improve the implementation so as to achieve better performance and more robustness.

4. SCRAMNet and HUMS: SCRAMNet (Shared Common Random Access Memory Network) is a system that is found to be suitable to implement HUMS. For this reason, we have conducted a simulation study to determine its suitability in handling the input data rates in HUMS. The system incorporates two important concepts: distributed and replicated shared-memory and insertion-ring network. Each node (also referred to as a station, or a host processor) on SCRAMNet has access to its own local copy of shared memory that is updated over a high-speed, serial-ring network.

   In order to analyze the performance of SCRAMNet in the structural health-monitoring environment, we have simulated the basic functionality of the system in terms of three major components---the links, the nodes (stations), and the data sources (sensors). Using the simulator, we measured the impact of factors such as the number of nodes, the load on the system, the error rate, and data filtering on message delay [4]. Figure 3, for examples, shows the effect of load on average delay.

   The results from this study are presented at the 2002 Western Multiconference [4].

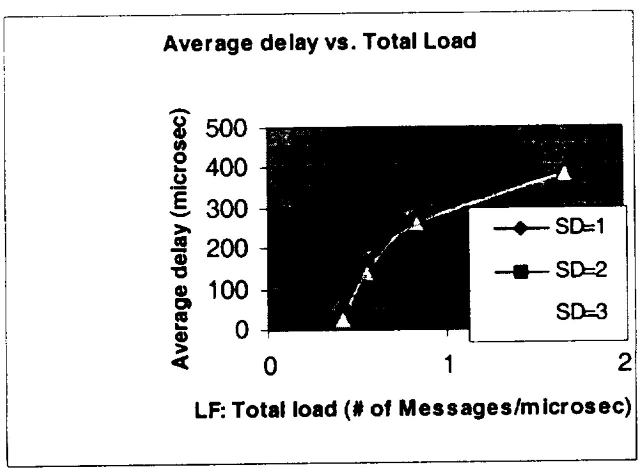


Figure 3. Effect of Load on Average Message Delay

5. **Architectural specification.** Architectural decisions have a great impact on the quality of software systems. When acquiring a large software system that will have a long lifetime within the acquiring organization, it is important that the organization develop an understanding of the requirements for such architectures. This understanding allows an organization to more formally specify its requirements as well as evaluate the candidate architectures. A formal software architectural evaluation provides several benefits including:

   - Allows the early detection of problems with the candidate architecture. It provides early insights into product capabilities and limitations.

- Allows for examination of the goodness-of-fit of the candidates to the functional needs as well as the performance requirements like reliability, scalability and maintainability of the desired systems.
- Formal specification and evaluation processes will force development organizations (vendors/bidders) to develop better and more suitable architectures (since they know the evaluation metrics ahead of time).

During the HUMS development and analysis process, we have realized that unless properly planned from the beginning, the task of evaluating the candidate architectures could become quite difficult or even impossible. Certainly, when we consider the complexity and criticality of HUMS, and the large number of candidate architectures that can be expected from the bidders (due to the size and importance of the project), the evaluation problem could be quite difficult. This also means that there is a potential for a wide range of specification formats, nomenclature, different degrees of details about the candidate architecture, etc. Sometimes the data needed for evaluation may be hidden in hundreds of pages of architectural descriptions provided by the bidders. In order to make the task of the evaluator much more effective, we have arrived at a combined specification and evaluation methodology. In other words, the specification methodology takes into account the evaluation process and accordingly decides on the specification formats. Once the candidate architectures are submitted in the prescribed format, the evaluation process becomes rather straightforward. Our findings are summarized in the attached document [5].

Publications (included in the appendix):

1. R. Mukkamala, "Distributed Scalable Architectures For Health Monitoring of Aerospace Architectures," Proceedings of 19th Digital Avionics System Conference, Philadelphia, PA, 7-13 October 2000.
2. R. Mukkamala, K. Bhoopalam, and S. Dammalapati, "Design and Analysis of a Scalable kernel for Health Monitoring of Aerospace Structures," Proceedings of 20th Digital Avionics System Conference, Daytona Beach, FL, 15-19 October 2001.
3. R. Mukkamala and K. Bhoopalam, "Building Robust Systems for Structural Health Usage and Monitoring," Proccedings of International Conference on Dependable Systems & Networks, Washington, D.C., June 23-26, 2002.
4. R. Mukkamala and M. Moharrum, "Modeling and Simulation of a Network for Structural Health Monitoring," Proceedings of Communications Networks and Distributed Systems Modeling and Simulation (CNDS 2002), San Antonio, TX, January 27-31, 2002.
5. R. Mukkamala, P. K. Anumula, K. Bhoopalam, and M. Moharrum, "Specification and Evaluation of Software Architectures for Health Usage and Monitoring Systems (HUMS)," Draft document (to be submitted for publication)

DISTRIBUTED SCALABLE ARCHITECTURES FOR HEALTH MONITORING OF AEROSPACE STRUCTURES

Ravi Mukkamala, Department of Computer Science, Old Dominion University, Norfolk, Virginia

Abstract

In recent studies, monitoring the health of certain aerospace structures has been shown to be a key step in reducing the life cycle costs for structural maintenance and inspection. Since the health of the structures ultimately determines the health of a vehicle, health monitoring is also an important prerequisite for improved aviation safety. In this paper, we present the preliminary results from our ongoing project on designing and evaluating architectures for integrated structural health monitoring. It has three major contributions. First, we present a survey of existing work. Second, we identify the characteristics of architectures for integrated health monitoring. Finally, we suggest evaluation metrics for a few characteristics.

Introduction

In recent studies, monitoring the health of certain aerospace structures has been shown to be a key step in reducing the life cycle costs for structural maintenance and inspection [1]. Since the health of the structures ultimately determines the health of a vehicle, health monitoring is also an important prerequisite for improved aviation safety. Much of the existing work in this area concentrates on using sensor fusion methods for monitoring single structures or components [2]. As the systems at the component level mature, there is a need to move these architectures from component level monitoring to system level monitoring.

One of the fundamental characteristics of the aviation sensor systems is that they are distributed across the physical structure of the aircraft. In addition, due to the high safety and reliability requirements placed on these systems, distributed architectures are a natural choice for these systems. Distributed architectures have the capability to offer the desired reliability and robustness in addition to other features such as scalability. However, they also have the capability of creating high communication overheads. In addition, they may introduce new bottlenecks such as the network bandwidth or synchronization requirements among distributed components that are otherwise absent in centralized systems. Hence, careful design and evaluation of the distributed architectures is necessary prior to adopting them for any health monitoring application domain.

In this paper, we present a summary of preliminary results from our ongoing project on designing and evaluating architectures for integrated health monitoring. Unlike other projects that deal with developing operational systems for a specific component, we concentrate at the higher architectural issues.

The paper is organized as follows. Section 2 gives a brief review of the existing work in this area. Section 3 describes different characteristics of the architectures. Section 4 illustrates evaluation measures for these architectures using two metrics ---scalability and robustness---as examples. Finally, Section 6 summarizes the current project status and describes future work.

Background

There has been considerable work in the structural health monitoring systems (SHMS). In this section, we survey some of the related work.

Much of the work in SHMS concentrates on modular but centralized decision systems. For example, the system developed by Kuduva et al [3] consists of sensors, local preprocessors, a central processor, and software for aircraft maintenance and logistical decisions. While the local processing and the modular structural offer flexibility to the SHMS, the central decision system could still be a bottleneck in scaling the system to different functions.

Openness also seems to be a favored feature for SHMS architectures [4]. The JAHUMS architecture emphasizes scalability and survivability [5]. It is based on open standards with as much COTS components as possible. Since they employ standard hardware and software interfaces, it is easy to insert new technologies or port to other systems. The open system concept is also adopted in the implementation of Integrated Mechanical Diagnostic system (IMD) [6]. The system is used to monitor the usage and health of helicopters. While software is well organized as modules, the hardware is centralized.

Another related area where much work has been done is distributed sensor management. Whenever multiple distributed sensors are involved in monitoring a structure or an object, it would be necessary to fuse output of several sensors to make a decision [7,8]. Lopez et al [7] present a three-level sensor agent architecture for the management of sensors in a distributed system. Here, the communication details are hidden in one level, while the task planning details are hidden in another. Finally, the task execution details are included in the third level. Clearly, the architecture provides a modular structure. However, the scalability issues are not so obvious.

In a distributed sensor architecture, one of the key issues is the distribution of sensor output to one or more processors that process (e.g., fuse, filter, etc.) the output. In terms of tools to support distributed processing in an aviation context, the Ring Buffered Network Bus or RBNB [9] is a good example. Here, a buffer is able to receive and cache input from external sources such as sensors. In turn, the processing units can read the data from the buffer in an asynchronous fashion. While the RBNB is more suitable for a geographically distributed system with data transferred using the TCP/IP, the concept must be equally applicable even for processing of sensors within an aircraft. Another tool that uses similar ideas for information distribution is Information Bus [10]. Like RBNB, the Information Bus is intended for geographically distributed processing elements. However, its underlying concepts such as the demand driven vs. event driven data flow, request/reply vs. publish/subscribe paradigms, and the idea of adapters for legacy systems are equally applicable for health monitoring system architectures.

Networking sensors using wireless communication is another area that is fast developing in both commercial and military environments [11,12]. In the context of commercial applications, wireless communication between sensors and detectors is used to track personnel, machinery, inventory, etc. Similarly, wireless sensor networks are used for terrain mapping in hazardous environments or during wars. Both systems are adaptable and scaleable. The wireless system concept coupled with sensors and computing elements resulted in a highly flexible and scaleable architecture. Sensor networking also provides robustness when the processing elements associated with the sensors can reconfigure among themselves whenever a fault is detected among the sensors.

Finally, in the area of architectural evaluations, there have been several contributions, mostly in the context of software architectures. But none of these directly address the SHMS architectures. In [13], the authors describe a comprehensive framework for evaluation of software architecture. They have developed COMPARE, an operational framework, to evaluate and compare architectures. The framework consists of several steps including eliciting objectives or goals of the evaluation, expressing the goals in terms of scenarios, qualitative and quantitative evaluation of the interactions (or impact) between the scenarios and the architectures, and a multivariable decision model to make final decision on the choice of the architectures. In [14], the authors discuss a scenario-based approach to evaluate software architecture. In this approach, scenarios are used to gain information about a system's ability to meet users' needs. Each scenario is expressed in terms of one or two sentences and describes a specific user expectation or need from the system. The scenarios are developed for all relevant stakeholders of the system under consideration. In this paper, we are interested in developing more quantitative evaluation metrics for the architectures.

Characteristics of Health Monitoring System Architectures

Before we start designing architectures or start developing architectural evaluation methods, we should first identify the characteristics of a health monitoring system mainly in the context of avionics systems. Since any such architecture consists of sensors, processors, interconnection networks, storage, and software, we characterize the architectures in terms of these components. Following is a list of some important characteristics.

- **Scalability.** One of the characteristics of the avionics industry is the diversity in size. In particular, the number of components that need to be covered by health monitoring systems changes from one aircraft to the other and from one customer to the other. In fact, the same customer, for the same aircraft, with a few components monitored today, may decide to expand the monitoring to several other components (also an extendibility characteristic). Similarly, the customer may decide to increase the number of sensors per component for increased reliability or fault-tolerance. Generally, such decisions are taken either due to reduction in the cost of technology or due to a need for increased safety. Any architecture on which the initial monitoring system is built should be scalable so that the additional sensors may be added to the existing system. Similarly, while the frequency of monitoring is at a low rate today, it may be increased tomorrow. The chosen architecture should be scalable so that it can handle the additional data and processing load. Certainly, this may require adding additional processing power or communication bandwidths to the existing system. The architecture should allow such scale-up to take place in the system.
- **Legacy Systems.** Due to high investment cost, aircrafts are being used for longer times [15]. Hence, several aircraft may need to be retrofit with the new health monitoring systems to reduce maintenance cost as well as offer improved safety and reliability. This means that the proposed architectures should be able to utilize the sensor and processing systems already in place in an aircraft. However, additional sensors, processors, or communication links, may be added for additional safety. This may require the development of some adapter software [10]. The architecture should allow such interfacing.
- **Openness.** While safety and reliability are the primary concerns of the aircraft industry, reduced costs are also a significant factor for survivability and profitability. Today, it has been well recognized that commercial off-the-shelf (COTS) components are much cheaper than proprietary custom-made software and hardware. More specifically, the architecture should have interface specifications that are fully defined, available to the public, and maintained according to some agreed upon standard [4,6,16].
- **Flexibility.** The ability to change or react with little penalty in time, effort, cost or performance is often referred to as flexibility. One of the key aspects of today's computing and communication technology is the continuous change. The same thing is true of sensor technology also. This means that any architecture that is closely tied to today's technology is likely to become obsolete in the very near future. Hence, we need an architecture that is easily adaptable to newer technologies, faster processors, more accurate data-rate sensors, etc. In addition, it should be able to incorporate newer sensor fusion and processing algorithms with higher CPU and I/O requirements, and tighter timing requirements.
- **Robustness.** Almost all avionics systems require high robustness to component failures due to the rigorous safety requirements. In fact, the architectures should be designed with the robustness in mind. Certainly, distributed systems have much higher potential to be robust than centralized systems. In addition, since there could be thousands of sensors in an aircraft, it is also likely that some of them may fail or malfunction during operation. However, such failures should not prevent the rest of the system from operating. In the case of replicated sensors, the architecture should be able tolerate the failure of a few replicas. Failure of a few data links or processors should not disable the entire system. In other words, fault tolerance as

well as fault isolation should be built into the architecture.

- **Extendability.** The ease with which a system or component can be modified to increase its storage or functional capacity is defined as extendability. Since the health usage and monitoring technology is at its infancy, it is likely to undergo several changes in the next few years. While the initial system have started with health monitoring of fuel tanks [17] and engines [6,18], the efforts to expand to other structures is well underway [4,19]. Hence, the proposed architectures should be amenable to extending the coverage to newer components, and to adding more accurate or reliable sensors. The architecture should allow newer sensor processing algorithms (implemented in hardware or software) to be added to the existing systems.
- **Intelligent monitoring.** A health monitoring architecture should be intelligent enough to distinguish between data and information. For example, an architecture that is simply a collection agent for sensor data will not likely have wide applicability. The user is more likely to be interested in overall health of the component rather than the individual data from each sensor. Similarly, suppose an abnormality has been sensed in a component, and it has been reported/recorded already. The sensor may continue to send the same data until the component is fixed, which occupies both communication bandwidth and storage yet adds very little value in terms of information. Similarly, ongoing collection of sensing data for a nominally operating component adds very little information to the data set. An intelligent architecture should be capable of filtering out, or reducing, such redundant data.

Metrics for Architectural Evaluation

Whenever we are presented with a set of health monitoring system architectures, we need to evaluate them with respect to the given objectives. There are more qualitative evaluation methods [16,20] in practice today than quantitative methods. Unfortunately, qualitative metrics are prone to individual subjectivity. For this reason, in our current work we concentrate on developing quantitative measures. Such measures are more rigorously defined and hence are more convincing and useful. In this section, we discuss ways to quantitatively define metrics that are otherwise used only in qualitative sense. Due to space constraints, we only discuss two metrics in detail.

In developing the metrics, we assume a health monitoring architecture to contain the following components.

- **Sensors (SN)** that generate signals (analog or digital) or data.
- **Processors (PR)** that process the data or signals to generate final and/or intermediate output
- **Communication links (or networks) (CN) that** carry data (or signals) from sensors to processors, processors to processors, processors to storage units, or storage units to processors. The nodes through which external users can access the data are also modeled as processors.
- **Software (SW)** that processes raw data and intermediate data (e.g., sensor fusion). Communication software is used for data communication.
- **Processes (PS)** that execute on processors to carry out the functionality of the software
- **Storage (ST) or I/O units** that store the observations or results.

(Since we are only dealing with a high-level view of the health monitoring system, we do not include the main memory or cache as components.)

We now use the model to develop quantitative metrics for evaluation.

Scalability. There are several definitions for scalability. An informal definition that is generally accepted in distributed systems is as follows. "A system or an algorithm is *scalable* with respect to a given parameter if its performance does not degrade drastically with the change in the parameter values within the given bounds [21]." In general, the term drastically is understood as "non-linearly." For example, consider the case of SMPs or Symmetric Multiple Processor systems [22]. Here, several processors share a common memory. Clearly, the system is not scalable beyond a certain number of processors as the shared memory becomes the bottleneck. Accordingly, increasing the number of processors will no longer produce the needed decrease in execution time due to increase in the number of processors. On the other hand, MPPs or Massively Parallel Processors [22], where each element has both processing and local memory and the elements are connected in an hypercube, are much more scalable since there is no shared memory. In the case of health monitoring systems, the main objective is to receive sensor outputs, process the outputs, and when necessary generate outputs either for further processing or for storing. Clearly, if there is a dedicated link between a processor and a sensor, then the communication link will no longer be a bottleneck. If a processor were to periodically poll a sensor for output, then the processor could become a bottleneck due to the overhead of periodic polling of several sensors that have been assigned to it.

Scalability analysis and bottleneck analysis are closely related to each other. In particular, scalability measures determine the *degree of matching* between a given computer architecture and the applications [22]. In our case, the sensor processing and storage algorithms are the key algorithms. They mainly involve processing and data communication activities. Following are seven quantitative definitions of scalability that could be used in this context.

SC1: If the number of sensors is increased from s to $a*s$, and to maintain the same overall response time (or some other performance measure) the number of processors are to be increased from p to $b*p$, then the scalability factor is defined as $SC1 = a/b$. Here, it is assumed that the average output per sensor as well as the processing speed remain unchanged. When SC1=1, it means that there is a linear relationship between the number of sensors and the number of processors. An alternate definition could be the ratio of incremental change in the number of sensors to the incremental change in the number of processors. Then $SC1 = \Delta s/\Delta p$. This measures the slope of the s versus p graph.

For example, if the number of sensors is increased from 100 to 120, and we had to increase the number of processors from 10 to 12, then $SC1$=1.2/1.2 or 1.0. This shows that there is a linear relationship between s and p. If we use the alternate definition, then $SC1$=20/2 = 10.0. In other words, for every increase of 10 sensors, we need to increase the number of processors by 1. In general, for any given architecture, the value of $SC1$ will vary over the range of input parameters, and will not be a constant over the entire range. In other words, the above values may only be valid at s=100, and need to be computed for other values of s also.

Depending on the relationship between s and p, we may also define other metrics that use other functions of s and p in computing SC1. For example, we can define SC1 as $\Delta(\log s)/\Delta(\log p)$.

SC2: If the number of sensors is increased from s to $a*s$, and to maintain the same overall response time the communication link bandwidth is to be increased from c to $b*c$, then the scalability factor is defined as $SC2 = a/b$. Here, it is assumed that the average output per sensor as well as the processing speed remains unchanged. Alternately, we can define it as the ratio of incremental change in the number of sensors to the incremental change in the interconnection network bandwidth. Then, $SC2 = \Delta s/\Delta c$.

For example, if the number of sensors is increased from 100 to 120, and we had to increase the link bandwidth from 2 Mbps to 2.2 Mbps, then $SC2$=1.2/1.1 or 1.09. Using the alternate definition, $SC2$=20/0.2 =100. As before, value of $SC2$ will over the range of its input parameters.

SC3: If the rate of output from each sensor is increased from x to $a*x$, and to maintain the same overall response time the processor speed is increased from y to b*y, then the scalability factor is defined as $SC3=a/b$. Alternately, it may be expressed as the ratio of the incremental change in number of sensors to the incremental change in the processor speed. Then, $SC3 = \Delta x/\Delta y$.

For example, if the output data rate per sensor increased from 100 bytes to 150 bytes, and we had to increase the processor speed from 20 MIPS to 25 MIPS, then $SC3$=1.5/1.25 or 1.2. Using the alternate definition, $SC3$= 50/5 =10.0.

Similar metrics could be defined using other factors such as the size and frequency of sensor outputs, the processing cost of sensor outputs, etc.

SC4: An alternate way to evaluate an architecture is to compare its performance against a reference or standard idealized architecture. For example, if $T_I(s,n,c)$ is the average response time (or some such metric where low is better) on the idealized architecture and $T(s,n,c)$ is the average response time on the given architecture, then scalability of the given architecture may be defined as $SC4 = T_I(s,n,c)/T(s,n,c)$. Larger the value of *SC4*, more scalable is the given architecture. (Here, *s*, *n*, and *c*, represent the number of sensors, the number of processors, and the interconnection network capacity, respectively.) This type of comparative approach is taken for comparing memory page replacement algorithms in operating systems [Silberschatz98]. Similar approach is taken while evaluating parallel system architectures [Hwang93].

SC5: Based on Gordon Bell's definition [Bell92], we can classify the distributed health monitoring architectures in terms of three categories: size-scalable, time scalable, or problem scalable.

A **size-scalable** health monitoring architecture is one that has a scaling range from a small to a large number of resource components---processors, interconnection network, and the storage. The expectation is to achieve linear improvement in performance as the size is increased linearly. For example, by doubling the number of processors we should be able to double its performance. Often, doubling the number of processors (or the processing speed) needs to be accompanied by corresponding increase in the bandwidth of the interconnection network. There may be a need to double the I/O bandwidth as well as the memory. Certainly, we need to take into account the cost, efficiency, and affordability in attempting to achieve size-scalability in an architecture. Using this definition, when the number of processors is increased from p to $a*p$, and we observe a performance improvement from q to $b*q$, then $SC5=b/a$. Here, the performance metric is such that high is better. We can define a similar metric when low-is-better metric such as response time or cost is used.

SC6: A **generation or time scalable** health monitoring architecture is one that can scale with the advances in component technology. So, when faster processors or faster interconnection links are available, we should be able to use the same architecture to build systems [21]. In general, all computer characteristics must scale proportionally: processing speed, memory speed and size, interconnection bandwidth and latency, I/O, and software overhead in order for an architecture to be useful for the health monitoring applications. For example, when processor speed is increased from x to $a*x$, and the performance (say high is better metric) is increased from y to $b*y$, then scalability may be defined as $SC6=b/a$. Alternately, we can use $SC6 = \Delta y/\Delta x$ definition.

SC7: A **problem scalable** health monitoring architecture is one that can work even when the number of sensors is increased. For example, if the number of sensors is increased from s to $a*s$, and the corresponding performance (where low-is-better) has changed from x to $b*x$, then $SC7=a/b$.

Clearly, the above three types of scalability---size, generation, and problem, are not completely independent of each other. Similarly, the resources---processors, interconnection networks, memory, and storage, are also not independent of each other. While evaluating an architecture, we need to take this fact into consideration.

Robustness. Let us now look at a second metric, robustness. Typically, robustness is defined as "The measure or extent of the ability of a system, such as a computer, communications, or data processing systems to continue to function despite the existence of faults in its component subsystems or parts." System performance, however, may be diminished or otherwise altered until the faults are corrected [23, 24]. This is a critical factor for the health monitoring architectures.

A health monitoring architecture is expected to have several functionalities. Some functionalities will be low-level functionalities such as ensuring that sensor data reaches its first level processor, or that a processor can write its data at its assigned storage unit. It may have some high-level functionalities such as delivering the status of a structure (probably computed by a sensor fusion function and stored on a disk) to an end user. There may be some intermediate functionalities such as ensuring that a processed sensor data is distributed to several processors executing some monitoring functions.

In defining the robustness of such architecture, we simply do not compute a single metric. Instead, we need to express it in terms of the offered robustness to different functionalities. Since we are referring to functionalities, there is an assumption here that we have made a mapping of the intended system to the architecture. For example, we can define the following metrics of robustness.

RO1: Probability with which a component's (e.g., engine's) status is recorded in the storage successfully.

Of course, to evaluate *RO1*, we need to know about the entire path from the sensors to the final storage unit, the reliability of different components on the path, the algorithms used along the path and their robustness, etc. So the computation would involve not only simple hardware reliabilities but also software and algorithmic robustness.

RO2: Probability with which a sensor algorithm is guaranteed to receive inputs from different sensors of a component within a given interval.

Metrics such as *RO2* are relevant to guarantee that the outputs from a sensor fusion algorithm are correct since its inputs are consistent (or timely).

RO3: Probability with which the status of at least k out of m ($m \geq k$) components will be available in the storage for the user.

To illustrate this metric, let us consider a case where a user needs a guarantee that at least two out of the following three components' status are always available: engine, rotor, and wings. Suppose the *RO1* metric for each of the three individual statuses is 0.95. However, an engineer needs to know the status of at least two of the three to determine the status of the overall aircraft. The probability with which the engineer can determine the aircraft status (in this case) is indicated by *RO3*. If the SHMS architecture provides completely independent systems for processing the three statuses, then RO3 would be 0.99. On the other hand, if the SHMS uses exactly the same system (i.e., data links and processors) from processing to storage of the three statuses, then RO3 would be 0.95. In all other cases, RO3 would be between 0.95 and 0.99. In other words, while *RO1* focuses at individual component level, *RO3* deals with subsystems and systems level status availability.

In general, in computing the robustness metrics, we need to model the architecture in terms of the reliabilities of the individual components and their interdependencies. In particular, more details are needed as to how a specific system is being mapped to an architecture, and the reliability of the individual components. Often, time guarantees (as in *RO2*) are to be combined with failures to determine the robustness of different functionalities.

Several metrics such as *RO1*- *RO3* may be defined for a specific application domain and for a given set of architectures. We are currently, in the process of defining such metrics at different layers of the health monitoring architectures.

Conclusion and Future Work

In this paper, we presented a summary of preliminary results from our ongoing research on development and evaluation of architectures for structural health monitoring. Here, we identified several characteristics of such architectures. Certainly, having distributed control and being scalable are the primary characteristics. In addition, features such as robustness, openness, and flexibility are also required. We have also illustrated how quantitative (rather than qualitative) metrics may be developed to evaluate these architectures. Scalability and robustness are used as examples for the illustration of the ideas.

One of the main objectives of our project is to develop tools that will help users in choosing appropriate health monitoring architectures for their specific applications. Currently, we are dealing with the characterization and evaluation issues. Once these issues are well understood, we will develop alternate architectures to test the developed evaluation methods.

Acknowledgements

The work is sponsored in part by NASA Langley Research Center under grant NAG-1-2279.

References

[1] Kent, R. M. and D. Murphy, 2000, *Health monitoring system technology assessments: Cost benefit analysis*, NASA Technical report. NASA/CR-2000-209848.

[2] Kim, H. M., T. J. Bartkowicz, and S. Smith, 1995, *Health monitoring of large structures*, Sound and Vibration, V. 29, pages 18-21.

[3] Boller, C. and C. Biemans, 1997, *Structural health monitoring in aircraft-state-of-thea-art.* Perspectives and benefits, Proc. International Workshop on Structural Health Monitoring, Stanford, USA, pp. 541-552.

[4] Harris, M. and D. Ngo, 1999, *An open architecture for next generation space onboard processing*, Proc. 1999 DASC, pp. 9.B.3-1 to 9.B.3-8.

[5] Haas, D. J., C.G. Schaefer, and D. Spracklen, 1999, *JAHUMS ACTD- A case study in open systems from a technology insertion perspective*, Proc. 1999 DASC, pp. 9.B.5-1 to 9.B.5-10.

[6] Muldoon, R.C., J. Gill, and L. D. Brock, 1999, *Integrated mechanical diagnostic (IMD) health and usage monitoring system (HUMS); an open system implementation case study*, Proc. 1999 DASC, 9.B.4-1 to 9.B.4-8.

[7] Lopez, J.M.M., F.J.J. Rodriguez, and J.R.C. Correderra, 1997, *Cooperative management of surveillance sensors*, Proc. IEEE Intl. Conf. Computational Cybernetics and Simulation, Systems, Vol. 1, pp. 845-850.

[8] Malhotra, R., *Temporal considerations in sensor management*, 1995, Proc. IEEE Aerospace and Electronics Conf., NAECON'95, Vol. 1, pp. 86-93.

[9] Freudinger, L.C., M.J. Miller, and K. Keifer, 1998, *A distributed computing environment for signal processing and systems health monitoring*, Proc. IEEE DASC 1998, Vol.1, pp. C35/1-C35/8.

[10] Oki, B., M. Pfluegl, A. Siegel, and D. Skeen, 1993, *The Information Bus---An architecture for extensible distributed systems*, ACM SIGOPS 1993, page 58-68.

[11] Estrin, D., R. Govindan, J. Heidmann, and S. Kumar, 1999, *Next century challenges: Scaleable coordination in sensor networks*, Proc. ACM Mobicom'99, Seattle, WA, pp. 263-269.

[12] Hopper, A., 1998, *Sensor-driven computing and communications*, IEE Colloquium on Living Life to the Full with Personal Technology, Digest No. 1998/268, pp. 21.1-21.4.

[13] Briand, L.C., S.J. Carriere, R. Katzman, and J. Wust, 1998, *A comprehensive framework for architectural evaluation*, ESE Report No. 46.98/E, Fraunhofer Institute for Experimental Software Engineering, Kaiserslautern, Germany, 8 pages.

[14] Kazman, R., G. Abowd, L. Bass, and P. Clements, 1996, *Scenario-based analysis of software architecture*, IEEE Software, November 1996, pp. 47-55.

[15] Boller, C. and C. Biemans, 1997, *Structural health monitoring in aircraft---state-of-the-art, perspectives and benefits*, Proc. International Workshop on Structural Health Monitoring, Stanford, USA, pp. 541-552.

[16] Bass, L., P. Clements, and R. Kazman, 1998, *Software architecture in practice*, Addison-Wesley.

[17] Melvin, L., et al, 1997, *Integrated vehicle health monitoring (IVHM) for aerospace vehicles*, Proc. International Workshop on Structural Health Monitoring, Stanford, USA, pp. 705-714.

[18] Dickson, B., J. Crokhite, S. Bielefeld, L.Killian, and R. Hayden, 1996, *Feasibility study of a rotorcraft health and usage monitoring system (HUMS): usage and structural life monitoring evaluation*, Army Research Laboratory, Report No. ARL-CR-90.

[19] Bartelds, G., 1997, *Aircraft structural health monitoring, prospects for smart solutions from a European Viewpoint*, Proc. International Workshop on Structural Health Monitoring, Stanford, USA, pp. 293-300.

[20] Hillard, R.F., M.K. Kurland, and S.D. Litvintchouk, 1997, *MITRE's architecture quality assessment*, Proc. 1997 Software Engineering and Economics Conferenece, pages 9.

[21] Coulouruis, G., J. Dollimore, and T. Kindberg, 1996, *Distributed systems: Concepts and design*, Addison-Wesley, Essex, England.

[22] Hwang, K, 1993, Advanced computer architecture: Parallelism, scalability, programmability, McGraw-Hill, Inc.

[23] Dhillon, B.S., 1987, Reliability in computer system design, Ablex Publishing Corp., Norwood, NJ.

[24] H.-Y. Wang and J. Rosenhead, A rigorous definition of robustness analysis. Pages 176-182. Journal of Operational Research Society, Feb. 2000, Vol. 51, No. 2, Mcmillan Publishers.

DESIGN AND ANALYSIS OF A SCALABLE KERNEL FOR HEALTH MANAGAEMENT OF AEROSPACE STRUCTURES

Ravi Mukkamala   Kailash Bhoopalam   Srihari Dammalapati

Department of Computer Science, Old Dominion University, Norfolk, Virginia 23529-0162

Abstract

In recent studies, monitoring the health of certain aerospace structures has been shown to be a key step in reducing the life cycle costs for structural maintenance and inspection [1]. Since the health of the structures ultimately determines the health of a vehicle, health monitoring is also an important prerequisite for improved aviation safety.

In collaboration with NASA Langley Research Center, we have been developing architectures for health and usage monitoring (HUMS) of aerospace structures [2]. In this paper, we describe a key component of the distributed architectures, the *HUMS kernel*. The contributions of this paper are three-fold. First, we describe the functionality of the kernel in terms of the services it offers to the layers above it. Second, we discuss several design issues of the kernel components. Finally, we analyze the kernel design in terms of its scalability.

Introduction

In recent studies, monitoring the health of certain aerospace structures has been shown to be a key step in reducing the lifecycle costs for structural maintenance and inspection [1]. Since the health of the structures ultimately determines the health of a vehicle, health monitoring is also an important prerequisite for improved aviation safety. The need for developing architectures for integrated structural health monitoring has been discussed in [2].

In collaboration with NASA Langley Research Center, we have been investigating some of the key characteristics of architectures for health and usage monitoring (HUMS) of aerospace structures. This resulted in a preliminary version of HUMS reference architecture (Figure 1). It is a layered architecture facilitating scalability, robustness, flexibility, and maintainability.

One of the key components of this architecture is the HUMS Kernel layer. The layer offers several basic services needed by a dynamic health monitoring system that are otherwise not available through a COTS operating system (or kernel). In this paper, we describe this layer in detail.

The paper is organized as follows. In Section 1, we describe the HUMS reference architecture proposed by our group. Section 2 describes the functionality of the HUMS kernel. In Section 3, we discuss several design alternatives for the kernel components. Section 4 discusses the scalability aspects of the kernel design. In Section 5, we give a brief outline of the current implementation, Finally, Section 6 has some final conclusions and future work.

HUMS Reference Architecture

The proposed reference architecture is shown in Figure 1. It is compatible with the Generic Open Architecture (GOA) proposed as a standard for avionics systems [3]. It consists of six layers that may be logically divided into three parts. The lower part deals with sensors and low-level processing and control. The middle part deals with system level processing and maintenance. Finally, the upper part is related to the application software and interfacing with the user.

The sensor-layer (hardware) consists of signal emitters (sensors) that are representative of physical

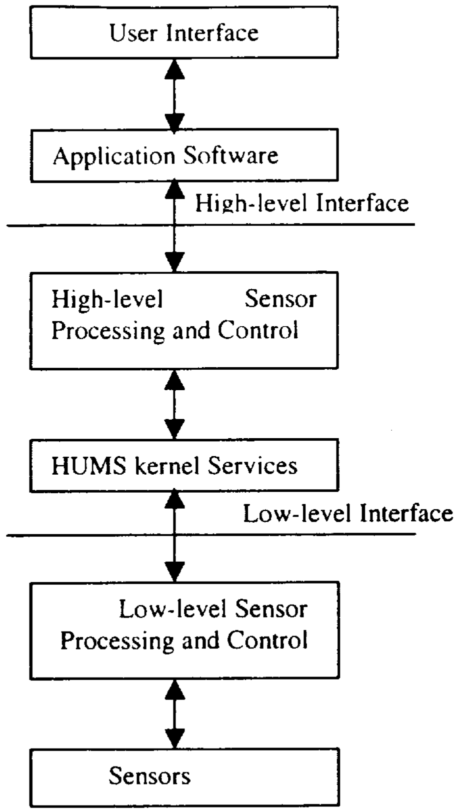


Figure 1. HUMS Reference Architecture

attributes such as temperature and pressure at a particular location (grid) on a structure.

The low-level sensor processing and control layer (predominantly hardware) would have the ability to interact (turn-off, turn-on, and fetch signal) with the sensors. It would also include any analog-to-digital conversion needed for some of the sensors. It may include some simple fusion algorithms implemented in hardware or software. In essence, the lower part of the architecture offers services related to sensor data and control.

The HUMS kernel contains the infrastructure components to provide a robust, dynamic, and maintainable distributed sensor system.

The high-level sensor processing and control layer has the capability to control (via low-level sensor controllers) groups of sensors. In addition, it offers some complex fusion and integration routines. HUMS kernel would constantly monitor the processes in this layer to enhance the reliability of the system.

The application software layer consists of several installation or domain specific software needed by the end-user.

Finally, the user interface offers interface to the end-user of the system. In case of HUMS, it could be the on-board staff such as the pilot or the ground staff such as engineers.

In this paper, we concentrate on the kernel layer of the architecture.

HUMS Kernel: Functionality

HUMS kernel offers the functionality needed to build a dynamic and robust distributed sensor system that is not usually offered by a COTS operating system. In particular, it offers services to support the following features.

- Dynamically add/delete/replace components such as sensors, services, processes, processors, and other system resources.
- Monitor resources and processes during their operation to detect any failures. It also has provisions to recover from process failures.
- Offer a hierarchical (i.e., logical) view of the system. This enables an application or an end-user to be transparent to low-level details (such as sensor Ids or number of sensors at a grid) and simply refer to high-level components (e.g., data values from all grids covering left-wing).
- Avoid strict synchronization between data producers (sensors) and data consumers (e.g., application/system processes). This enables to build more flexible systems.

In general, it is designed with the objective of supporting scalability, robustness, and flexibility in HUMS. In this section, we describe the basic services offered by the kernel.

A block diagram of the HUMS kernel that we have designed and implemented is shown in Figure 2. The kernel software is organized into five basic modules: lifecycle services, naming services, relationship services, buffering services, and monitoring & relocation services. We will now discuss the functionality of each of the modules in detail.

Lifecycle Services

HUMS is a dynamic system in the sense that new sensors may be added when need arises, additional components may be covered by sensors, and failed sensors may be replaced or removed. In addition, there may be a need to regroup sensors or replace them when newer technologies become available. The lifecycle services module is responsible for managing these changes. Basically, it keeps track of the lifecycle of a sensor, a component, or a service from the time it is first added, configured, and used to the time it is ultimately replaced or removed from the system. It interacts with the relationship services to store and retrieve information about the sensors and other components in the system. Similarly, it interacts with the naming services to obtain an internal name when a new component (or sensor) is added to HUMS.

This module is also responsible for initially starting processes (both service and kernel) and for restarting failed processes. It interacts with the monitoring and relocation service in monitoring the processes that it is responsible for. In case it detects failure of a process, it interacts with the cloner component of the monitoring & relocation service to restart the process.

In addition, it may keep track of the utilization and state of system resources (e.g., processors and network links) to determine where a process may be executed or if it needs to be migrated to another process for better performance.

Relationship services

This service provides the relationships among various structural components (e.g., engine, left-wing, right-wing, etc.) and the corresponding HUMS elements (e.g., sensors, sensor-controllers, processes, etc.). It provides an interface to store and retrieve the relationship information. Suppose a user (using an application program) wants to retrieve all sensor values related to a specific component such as a left wing, the application program first needs to access the relationship service to identify all sensors related to that component. It can then access other components to access the data. In summary, it is a database of all

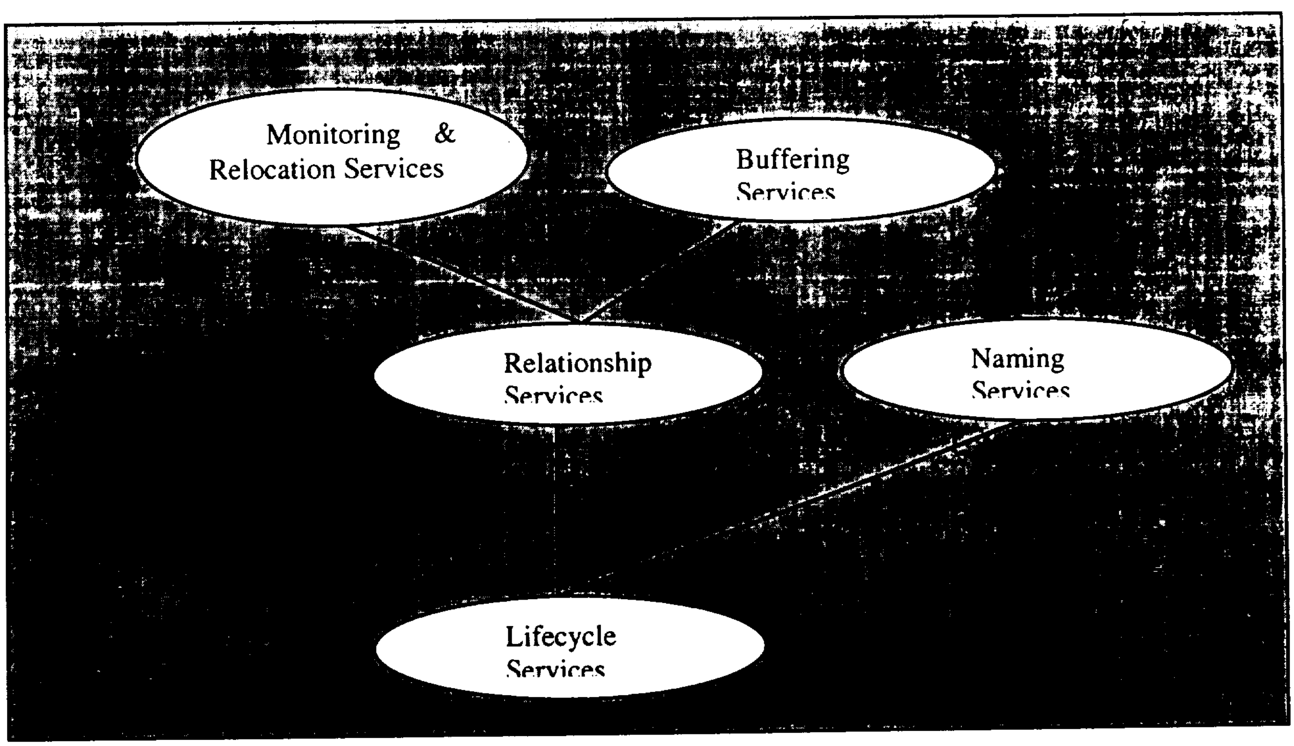


Figure 2. HUMS Kernel: A block diagram

types of relationships and associations in the HUMS system.

Naming Services

With thousands of sensors and hundreds of components in a typical HUMS system, it is necessary to assign structured names (e.g., hierarchical as in an IP address) for them. Naming service generates internal names (or addresses) for the sensors and components (and subcomponents). A name is provided as and when the lifecycle service requests for one providing all the necessary details.

Buffering services

In a signal-driven system such as HUMS, sensors are the elementary signal *producers*. Of course, there may be several other processes (e.g., virtual sensors) that may process several physical sensor values and generates a virtual sensor value to be used by other high-level processes. Similarly, there are several processes that act as consumers of the generated signals (or data). In fact, one or more consumer processes may consume signals from a single sensor. In addition, the speed and timing of the producers may not be synchronous with the consumer processes. The buffering services acts as the interface between producers and consumers. Hence, both the producers (e.g., sensors, virtual sensors, and sensor-controllers) and the consumers (high-level processes) access the buffering services. In addition to current data, it also stores historical data.

Monitoring and Relocation Services

This module implements the basic features needed to offer robustness in HUMS. Once this module has been assigned the task of monitoring a process (by registering the process), it periodically checks the status of that process. It can check both for liveness as well as for correct functionality. In case a process is found to be faulty, it kills the process and attempts to start another copy of the process at the same processor. In case of a processor failure, it will attempt to relocate all its processes to one or more other processors. It makes use of a cloner sub-module for the relocations and restarts.

HUMS Kernel: Design Aspects

So far, we have discussed the HUMS kernel as a single entity consisting of several components. However, unlike the traditional operating system kernels that need to be implemented as single entities at all the nodes in a system, pieces of the HUMS kernel can be implemented at different nodes. Some of the nodes may have all the services, and some may need just a few. In fact, as discussed below, each component itself may be implemented in a distributed manner among a set of processors. Following is a discussion of these components and some design choices in their implementation. The stated choices are by no means exhaustive.

Lifecycle Services

This is an essential module to start processes as well as restart them in case of failures. One of the primary components of this module is the *process controller* that exists either in each processor or in a set of processors with a unique address. The process controller starts processes (system and application) at a processor at system startup. In addition, whenever a process is killed, it restarts it with the available executable code.

In addition to these basic functionalities, the lifecycle services module may include the following features:

- Monitor resource (e.g., CPU, memory, and communications) utilization of processes to check if they are exceeding the expected values.
- Monitor the load on different processors and if necessary redistribute the load.
- Request copies of executables from cloner when local copies are corrupted.
- Allow for processes from other processors to be migrated to local processor(s).
- Allow services (processes) to be added, deleted, or modified as the system evolves.

In a simple configuration, a lifecycle module executing in one processor is independent of lifecycle module executing at another processor. In a more complex system, the lifecycle component could be implemented as a distributed service where lifecycle modules at different processors act as a single service. We have not investigated the latter configuration in detail.

Relationship services

This is essentially a database of relationships among various structural components and HUMS components. This service is frequently used by the lifecycle services to store and retrieve relationships. In addition, most application software needs to access this service for converting the logical view of a user to the physical view of the HUMS system. Several design choices are possible in implementing this module.

- A simple choice is to implement it at a single processor (the centralized option). In this case, all other kernel modules need to access this service only at this processor.
- A more robust approach is to replicate the module at a set of selected processors. Each module would then contain a copy of the entire relationship database. Other kernel modules can access this service at any of the replicated locations.
- A more practical approach is to let a group of processors that are either geographically or logically related share a relationship module that only contains data related to the processes, components, and sensors connected to these processors. If necessary, the partial database could be replicated at a few processors in a cluster.

This module could receive requests such as "retrieve all sub-components of the left-wing" or "retrieve all grid-ids in a sub-component of the left-wing" from high-level processes.

Naming Services

This module generates unique identifiers (Ids) for all components, sub-components, services, and sensors in HUMS. This is similar to services such as a file system generating internal names to user files in a typical computer operating system. It is only called by the lifecycle services whenever a new entity is to be entered into the system. As in the case of the previous modules, there are a few choices for this module also.

- A centralized naming service executing on a single processor in the system.
- Autonomous naming service module executing in different parts of the HUMS system. For example, a group of related components might share one name server.
- Wherever there is a lifecycle module, a naming module also may be implemented.
- For robustness, we may consider replicated naming servers.

In general, the location and design choices for naming module may be closely related to the choices for lifecycle module.

Buffering services

This is a key component of the HUMS kernel. As it is used in storage and retrieval of data generated by the sensors, this is also the most frequently accessed module under normal operations

This module consists of three main components: buffer controller, data server, and file system. The buffer controller acts as the interface to the buffering services. Other processes interact with the controller to store and retrieve data. The buffer controller sends data to the data server. Data severs cache the data received from the controller (initially generated by the sensors and virtual sensors). They then send historical data to the file systems for permanent storage.

One possible way to store and retrieve information is discussed here.

The sensors (S) generate data and send it to sensor-controllers (SC). The sensor-controllers convert the signals to digital form, optionally do some preliminary analysis (e.g., fusion), and send it to the buffer controller along with identifying information such as sensor identification, timestamp, etc. The buffer controller (BC) receives the data and forwards it to one of the data servers (DS) for buffering. The data servers buffer the data and forward the data to the file systems for permanent storage.

Similarly, when a process wishes to retrieve data, it first accesses the buffer controller that provides the process with the location of the data server of the requesting data. The process then accesses the data through the data provider component of the data server. The request may be finally serviced by data residing in a buffer or a file system.

There are several design choices for implementing this module.

- Centralized versus distributed buffer controller. In the centralized case, a single buffer controller serves all processes in HUMS. While this is simple to design and implement, it may not be suitable due to the distributed nature of HUMS. The distributed controllers could be either autonomous (several buffer controllers acting independently) or controllers acting as agents for a system-wide service.
- The data servers could also be centralized or distributed. But typically, due to the distributed nature of the data generators (sensors), the distributed option is a natural choice. Whether data servers are located close to the data producers or the data consumers is a design choice and depends on specific application domain. In any case, each data server is autonomous.
- The file servers could also be centralized or distributed. Once again, for robustness, load balancing, and reduced communication load, it is preferable to distribute the file servers. These are truly autonomous units.

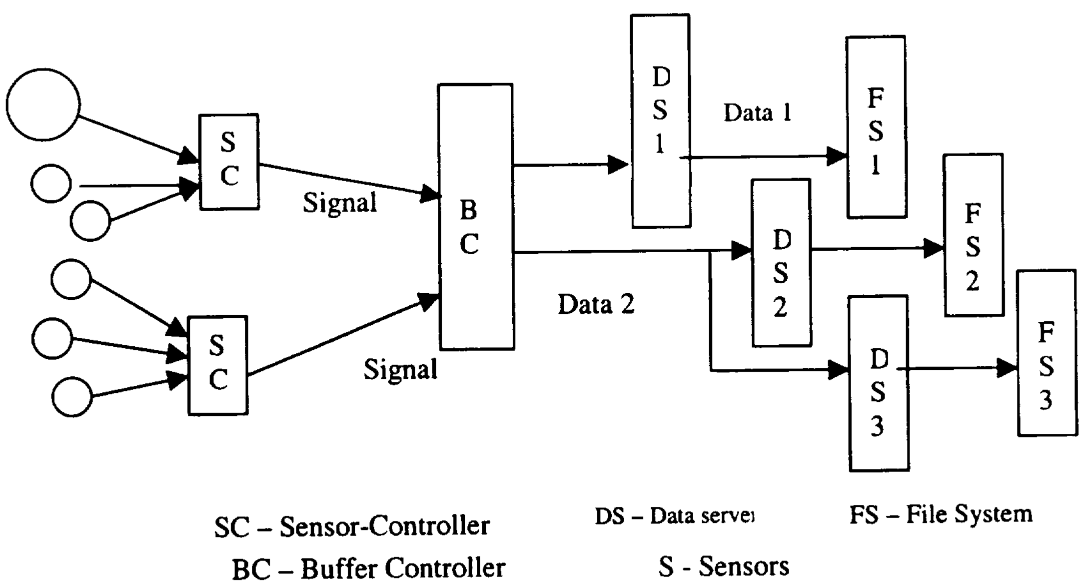


Figure 3. Data storage using the Buffering Services

- For fault-tolerance and load balancing purposes, a buffer controller may send the same data to one or more data servers. Each data server may send the data to one or more file servers.

Since the buffering services module is the most frequently used module, servicing data storage and retrieval requests, it is one of the performance bottlenecks in the system. For this reason, its design and implementation has to be tuned to the specific application needs.

Monitoring and Relocation Services (MRS)

This module provides fault-tolerance for essential HUMS components and services by the use of polling and cloning. The *poller* is the main component of the monitoring system. A process to be monitored needs to be registered with the monitoring system. In addition, the process should also contain a *listener* component. The poller polls all the processes in its list at regular intervals. Since the monitoring process is not only concerned about the status (live or dead) of a process but also about its behavior (normal or faulty), the process should maintain information about its behavior in some variable. One mechanism to measure the normality of behavior is the execution time. So a process could maintain the time it took for executing a specific chosen function in some variable that it shares with the listener component. Thus, when the poller polls a process, its listener component would reply with the value in this variable. Since the normal range of values of this variable is provided a priori to the poller (at the time of registration), the poller can decide whether or not a process is behaving normally. When this value exceeds the specified range consecutively several times, the poller declares the process as being faulty and sends a kill signal. On receiving the kill signal, the listener kills the process. When a process is killed, all resources held by it are also released.

On noticing that a process has been killed, the local process controller will attempt to restart it again. The poller once again attempts to poll it. In case a poller cannot establish a connection with a registered process, it makes a note of it in its log.

The second component of interest in MRS is the *cloner*. The cloner maintains copies of executable code of all processes registered with the MRS. Whenever a process controller fails to start a process or a process needs to be started at a new processor, the process controller contacts the cloner. The cloner then supplies the executable to start a new process.

The third component is the *redirector*. Whenever a process fails (or is killed), a new process is restarted. The new process could also be restarted at a new processor. The processes that were in communication with the earlier process now need to establish a connection with the new process. But they need to know the process identification (ID) and the processor address (e.g., IP address) where the new process is executing. Since a process controller had earlier supplied this information to the redirector component, it can provide it to the requesting process. After getting the process ID and processor address, the requesting process can reestablish a connection with the restarted process. There may be other mechanisms to achieve the same.

In essence, the poller, the cloner, and the redirector components achieve the objectives of the monitoring and relocation services module.

Now let us briefly look at the design alternates we may have in terms of the distribution of MRS in the system.

- Certainly, a centralized MRS is not practical even for systems with ten processors. So we need to implement it as a distributed service. Most likely, several autonomous MRS modules implemented in the system (e.g., one per group of processors) would be a practical solution.
- Since MRS contains the three components discussed above, another design choice is about the distribution of these components. One choice is to have all the three components as a unit executing at one processor where the MRS is to execute. However, this is

not an absolute requirement. In fact, it may be desirable to separate the cloner component from the rest since it may require large disk space. We could also have a few cloners distributed among the processors in a system. These should be accessible to all process controllers in the system. There could be some replication among the cloners (i.e., the code for a process is available at more than one cloner) so that fault-tolerance can be incorporated at this level.

- The poller should certainly be limited to poll only a few processes on the designated processors. Thus, we may have many more pollers in the entire system.
- The redirection components are needed only in the case of process failures. Thus, only a few of these are needed in the entire system.

In essence, while we logically consider MRS as a single module, the actual implementation may treat the three constituent components as independent units.

To illustrate the usage of the kernel, let us consider the case of adding a new service to HUMS. The example is described in terms of the type of the request, the pre-conditions, necessary steps to execute, and the post-conditions.

Request: Add a new service
Pre-conditions:

- The user has a new service to add.
- The user knows the processor ID where the service is to be added.

Steps:

- The user specifies the name and path of the executable that accomplishes the service.
- The user specifies the processor ID where the service will be made available.
- The HUMS kernel verifies the status of the resident processor. The system transfers the executable to the process controller where the service is going to execute. The process controller starts the process.
- The process is registered with the monitoring and relocation service and a copy of its executable is supplied to the cloner.
- The name of the service and its relationship with other components is recorded with the relationship services.

Post-condition:

- The service is now available at the specified processor.

HUMS Kernel: Scalability

One of the characteristics of the avionics applications is the diversity in size. In particular, the number of components that need to be covered by health monitoring systems changes from one aircraft to the other. In fact, the same aircraft with a few components covered currently may decide to expand the monitoring to several other components in future. Similarly, the number of sensors covering a component may be increased for robustness. All these factors demand a scalable HUMS kernel to manage the increased load on the system.

We have designed the HUMS kernel with the scalability and robustness in mind. To justify our claim, we give a list of design decisions that lead to this conclusion.

Buffering services is one of the key components of the kernel. This is also likely to be a performance bottleneck if not properly designed. The following design decisions enable this component to support scalability.

- The service is designed to be implemented as autonomous units in different clusters. Accordingly, even when the number of sensors and/or the number of structural components to be covered is scaled up, the load can be handled by establishing additional service units.
- Since each of the buffering service units can incorporate multiple data servers supporting buffers and file systems, it is easy to make a unit more scalable by increasing the number of data servers. If the buffer controller is found to be a bottleneck, using the relationship services, it is easy to

reassign the sensors (and sensor-controllers) to different buffering services.

- Since the buffer controller, the buffers, and the file servers are not closely tied in terms of geographical proximity, there is much more flexibility in assigning resources when a system is scaled up.

Let us now look at another key component, the monitoring and relationship services (MRS). This is also designed keeping scalability in mind.

- MRS is also designed so that autonomous service units may be established in the system. For example, one cluster of processors may have an autonomous MRS unit dedicated to processes executing in the cluster. As a consequence, when newer structural components are to be added due to scaling up, new MRS units can easily be established without increasing the load on the existing units.
- There is a single polling unit per MRS covering all processes in that unit. However, since it is not a computationally intensive task, it should be able to scale up when the number of processes to be monitored is increased. The communication load, however, may be significant. In case, the system is scaled up in terms of the number of monitored processes, then it is easy to install a new MRS unit and reassign the processes among the units.
- The cloning component may be implemented as autonomous units, each unit containing the executable code of several processes. In case, the number of processes in a system increases, additional cloners could be installed and reassign processes to cloners. The relationship service and the lifecycle service may be used for the reassignments.

The lifecycle service is another key component of the HUMS kernel. It is only needed when starting new processes/services or restarting failed processes. In that sense, it is not a bottleneck under normal operations. However, when system failures are detected, it determines the system performance during recovery. This module is also designed so it can be implemented as autonomous units. Each processor in the HUMS system has a lifecycle component, the process controller. Each process controller is autonomous. Thus, when a system is scaled up and incorporates additional processors, we simply need to initiate additional process controllers at the processors. To monitor other system resources such as the processors, we can incorporate autonomous units (say, one per cluster) to reassign processes under heavy loads. Thus, in a scaled up system we need to increase the number of such monitoring units to meet the higher demands.

The relationship service is also designed with scalability in mind. For example, each cluster can have its own relationship service module storing related relationships. Thus, system scalability can easily be achieved by adding additional relationship service modules.

The naming service has a role to play only when a new component/service is first introduced in the system. Since this is not a routine operation, this module can never be a bottleneck in achieving system scalability.

HUMS Kernel: Implementation

We have implemented a preliminary version of he HUMS kernel on a Unix platform. We have implemented both a centralized system version and a distributed version. In the centralized version, there is a single processor to which all sensors are connected. The sensors are simulated by means of data generating processes.

In the distributed version, several processors in the system are connected using Ethernets. TCP/IP is used as a means of inter-process communication.

In both cases, the relationship service is implemented using an Oracle database. Alternately, it could have been implemented as a simple file service also. For the administrator to interact with the lifecycle services, to add/delete services/processes/sensors, a Java-based GUI has been implemented.

The sensors have been simulated by processes that emit signals at random (0-20sec) intervals to the sensor-controller. The communication between the sensor and the sensor-controller is implemented using UDP [4]. UDP was the chosen protocol because it uses *best-effort* delivery and does not impose the overhead of handshaking. This is similar to the real environment, where sensors produce signals irrespective of whether the communication medium is able to accommodate them or not. Additionally the sensors are not concerned about the signals reaching the destination.

The sensor-controllers (SC) receive the signals from the sensors over a 20 second period. Each SC performs a simple sensor fusion (averaging) of the signals from each grid and transmits the signals to the buffering service that is located at a well-known address.

The SCs transmit the signals to the buffering service in intervals of 20 seconds. The buffering service is located at well-known addresses (i.e., the ip and port of the buffering service is known to the sensor-controllers).

The buffering service is duplicated. A primary buffering service and a secondary buffering service run at different machines and both their addresses are known to all the sensor-controllers. The communication between the sensor-controllers and the buffering service is achieved using TCP [4]. A failure of the sensor-controller to send signals to the buffering service would be interpreted as the failure of the buffering service. When such a failure occurs, the sensor-controller sends signals to another buffering service. The secondary buffering service is configured to monitor the primary. When the primary is alive the secondary remains dormant, but when the primary dies, the secondary becomes active. The primary has the highest priority to handle signals, hence if the primary is restarted again, it becomes active and the secondary shifts to the passive mode. Thus, the communication protocol between the sensor-controllers and the buffering service is provided with sufficient intelligence to determine failure of transmission of the signals.

Currently, we are discovering means to improve the implementation so as to achieve better performance and more robustness.

Conclusion and Future Work

In this paper, we have summarized some salient features of the HUMS kernel that we designed as part of a project to develop HUMS architectures. The kernel, with its five modules, is shown to be scalable---it can function even when the number of structural components, the number of sensors, and the number of processes is increased. The scalability is mainly achieved using autonomous and distributed kernel components. Robustness is achieved using replication of data and processes. A preliminary version of the kernel has now been implemented. We are currently in the process of studying its performance and finding means to improve it.

Acknowledgements

The work is sponsored in part by NASA Langley research Center under research grant NAG-1-2279.

References

[1] Kent, R. M. and D. Murphy, 2000, *Health monitoring system technology assessments: Cost benefit analysis*, NASA Technical report. NASA/CR-2000-209848.

[2] Mukkamala, R., 2000, *Distributed scalable architectures for health monitoring of aerospace structures*, Proc. IEEE DASC 2000, pp. 6.C.4.1-6.C.4.8.

[3] *Generic open architecture (GOA) framework*, Society of Automotive Engineers, Inc., AS4893, 1996-01.

[4] Tanenbaum, A. S., Computer Networks, Third Edition, Prentice Hall, PTR, 1996.

Building Robust Systems for Structural Health Usage and Monitoring

Ravi Mukkamala Kailash Bhoopalam
Department of Computer Science, Old Dominion University
{mukka, kbhoopal}@cs.odu.edu

1. Introduction

In recent studies, monitoring the health of certain aerospace structures has been shown to be a key step in reducing the life-cycle costs of structural maintenance and inspection [1]. Since the health of structures ultimately determines the health of a vehicle, health monitoring is an important prerequisite for aviation safety. Due to the safety and criticality of aviation systems, it is important that the health and usage monitoring systems (HUMS) be quite robust [2].

In collaboration with NASA Langley Research Center, we have been designing a prototype HUMS. Since flexibility is one of the desired features of these systems, we have designed a kernel to support much of the system functionality needed by different applications [3]. In this paper, we describe our on going effort in designing and prototyping a HUMS kernel to support robust and critical health monitoring systems for aviation applications.

. In a typical HUMS system, sensors generate data (periodically/aperiodically) and send data to sensor controllers. The monitoring system should reliably capture and process the data, and inform higher level processes of any anomalous behavior. We have explored the following three techniques to achieve robust HUMS.

- **Process and Data Reliability**. This is used to identify/isolate/rectify software failures and signal data. This reliability is provided by the HUMS kernel services. The kernel services periodically monitor key processes and provide mechanisms to correct failed processes.
- **Component Reliability**. This is achieved by identifying/isolating hardware and sensor failures from disrupting the functionality of the system. Component reliability is also ensured by distributing and replicating important components of the system and by ensuring proper filtering and isolating mechanisms to prevent Byzantine failures from reducing the functional effectiveness of the rest of the system.
- **Communication Reliability**. This is achieved by identifying the protocols/connectivity mechanisms that can be applied under various scenarios to ensure reliable communication among the various components of the system. Depending on the interactions among the components within a layer, the reliable communication protocols may be different in different layers of the system.

In the rest of the paper, we briefly describe the HUMS kernel and its reliability mechanisms.

2. The HUMS Kernel

The HUMS kernel is a layer that acts as an interface between the low-level sensors and sensor controllers and the high-level user and application processes. Its services include, but not limited to, the following:

- Reliability enhancement and Fault-tolerance mechanisms (i.e., make the sensor/sensor-controller failures transparent to higher layers)
- Bookkeeping (buffering and writing to a stable disk to isolate higher layers from dealing with flow-control problems associated with data generators)
- Dynamic system management (add/delete components or processes dynamically at run time)
- Data querying at different levels of components and component aggregation

The kernel is organized into five modules based on the services they offer: Monitoring and relocation services, buffering services, life-cycle services, relationship services, and naming services. Among these five, only the first three are relevant for this paper since they support robustness.

2.1 Monitoring and Relocation Services

The monitoring and relocation services can be provided with the following capabilities to improve the reliability of the system. It can be configured to:

- Monitor the *aliveness* of key processes.
- Detect erroneous/faulty processes by comparing available results from the processes with sets of expected results for various input data. This may be accomplished by remotely initiating self-diagnostic procedures on key processes and by comparing the actual results of the diagnostics with expected results.

- Detect failed processors and identify alternate compatible processors to migrate key processes from the failed processors.
- Interact with the lifecycle services to communicate erroneous processes and provide the lifecycle service with appropriate information to start/kill/restart new and existing processes.
- Prevent cascading failure of processes by allowing processes to look for alternate sources of data when their current data sources (could be a process) are not available.

This module interacts with the lifecycle services in implementing the above reliability mechanisms.

2.2 Buffering Services

The buffering service provides the following services.

- It stores the signal data at multiple locations thereby enhancing data availability. Such a mechanism is advantageous if data in a buffer is corrupted, or if an access path to a buffer is broken.
- It acts as a "data request broker" among multiple producers of signal data and multiple consumers of signal data that may not be synchronous in their production or consumption of data. This facility eases data access because consumers of data are not required to be aware of the location of data.
- It makes the data available to the consumers, closer to the point of consumption. Such a mechanism reduces demand latency.

2.3 Lifecycle Services

The lifecycle service is also called the process controller because it is provided with the capability to start and kill services (processes). The lifecycle service could provide one or more of the following services depending on the reliability requirements of the system.

- It can act as a booting program to start processes in sequence.
- It can monitor the CPU utilization of various processes to obtain a snapshot of the system. This snapshot can be compared against normally expected system states. Any anomalies can be recorded for further analysis, or the process controller can take pro-active steps (if it is provided with sufficient information) to kill or restart erroneous processes.
- The process controller can interact with the monitoring and relocation service to improve the reliability of the system.

Life cycle services can be distributed in different subsystems, and can vary in complexity depending on the critical nature of the processes in its subsystem.

3. Reliability Enhancements at Other Layers

The reliability of HUMS can be further enhanced through mechanisms offered at other layers (besides the kernel layer).

For example, at the sensor layer, if sensors are purely data emitters, the only method of providing reliability is by replication of sensors. Alternatively, sensors can be equipped with data-link protocols and buffers to react to retransmission requests and to store small amounts of data. This would allow low-level sensor controllers to request for data that could have been lost during transmission.

Similarly, the low-level sensor-controller functionality may be enhanced to improve overall reliability. For example, the sensor fusion algorithms could takes into consideration the non-availability of sensor data, Byzantine signals (signals from truant sensors), and sensor priorities (weights assigned to sensors depending on their reliability or accuracy). This layer can also provide simple error detection or correction facilities by padding sensor data with parity bits, CRC, sequence numbers, etc. Complex protocols that enable acknowledgements and retransmission can be used to interface this layer with the buffering service. This would minimize transmission losses of data.

4. Conclusion

In this paper, we have briefly summarized our ongoing work on building robust distributed structural health usage and monitoring systems. In particular, we have described the mechanisms used in the kernel layer to support robust systems. We are currently working on enhancing these mechanisms.

5. References

[1] Kent, R. M. and D. Murphy, 2000, *Health monitoring system technology assessments: Cost benefit analysis*, NASA Technical report. NASA/CR-2000-209848.

[2] Mukkamala, R., "Distributed scalable architectures for health monitoring of aerospace structures," *19th Digital Avionics Systems Conference (DASC'00)*, Philadelphia, PA, October 7-11, pp. 6.C.4.1-6.C.4.8, 2000.

[3] Mukkamala, R., K. Bhoopalam, and S. Dammalapati, "Design and analysis of a scalable kernel for health management of aerospace structures," *20th Digital Avionics Systems Conference (DASC'01)*, Daytona Beach, FL, October 15-19, pp. 3.D.2.1-3.D.2.10, 2001.

Modeling and Simulation of a Network for Structural Health Monitoring

Ravi Mukkamala Mohamed Moharrum
Department of Computer Science
Old Dominion University
Norfolk, Virginia 23529-0162
{mukka,mohar_m}@cs.odu.edu

Keywords: HUMS, performance evaluation, SCRAMNet, sensor networks, structural health-management.

Abstract
Monitoring the health of aerospace structures is an important step in reducing the life-cycle costs of maintenance for aircrafts. This is mainly achieved by using a network of sensors distributed among the structural components of a vehicle. The output from the sensors is then collected and processed by several higher-level processes to determine the structural health. In this paper, we report the results from our study of determining the suitability of SCRAMNet, a distributed shared memory network, in supporting the data distribution functionality of a sensor network for structural health. We have developed a simulation model of SCRAMNet and used it to evaluate the effectiveness of implementing a sensor network on top of SCRAMNet. The performance of the system is primarily measured in terms of message delay. The sensitivity of the system to the number of nodes, the error rates, and data filtering factors is also reported.

INTRODUCTION

In recent studies, monitoring the health of certain aerospace structures has been shown to be a key step in reducing the lifecycle costs for structural maintenance and inspection [1]. Since the health of the structures ultimately determines the health of a vehicle, health monitoring is also an important prerequisite for improved aviation safety. The need for developing architectures for integrated structural health monitoring has been discussed in [2]. In collaboration with NASA Langley Research Center, we have been developing architectures for health and usage monitoring system (HUMS) of aerospace structures [3].

One of the key ingredients of HUMS is the availability of a network that can propagate the sensor values to all the monitoring processes in a timely manner. The monitoring processes are themselves distributed across the system and probably at a ground station also. In addition, since a sensor may be generating the same data (e.g., unchanged temperature values), it would be effective to use a system that can filter the unchanged data from being unnecessarily transmitted. Of course, the applications should also be designed taking this into consideration. SCRAMNet (Shared Common Random Access Memory Network) [4,5] seems to embed many of these features. It is also used in several real-time applications [4,7].

In this paper, we study the suitability of SCRAMNet in supporting the needs of HUMS. The paper is organized as follows. In section 2, we provide some background information on SCRAMNet and HUMS. Section 3 provides a description of the simulation model adopted in our study. In section 4, we discuss the results obtained from the simulation studies and their implications. Finally, in section 5, we provide some conclusions and discuss future work.

BACKGROUND

In this section, we provide the background for SCRAMNet and HUMS. Only the background essential to understand this paper is presented. Interested readers may refer to the references provided for additional information.

SCRAMNet [4,5,6,7,8]

SCRAMNet (Shared Common Random Access Memory Network) incorporates two important concepts: distributed and replicated shared-memory and insertion-ring network [4, 9]. Each node (also referred to as a station, or a host processor) on SCRAMNet has access to its own local copy of shared memory that is updated over a high-speed, serial-ring network. To utilize the replicated shared-memory concept, distributed processes executing at a node map their global data structures into the dual-port memory located at the node [4,5]. Any time an application process updates a data structure located in its local SCRAMNet memory, the memory address and its contents are immediately (and automatically) broadcast to all other nodes on the SCRAMNet network. This automatic data transfer requires no software intervention or backplane loading, enabling the host computer to provide more processing resources to the applications [4].

HUMS [1,2,3]

In a typical HUMS (Health Usage and Monitoring System), sensors are placed within the structural components of an aircraft (the system being monitored). The sensors could be analog or digital, generating periodic or aperiodic output [1]. For example, a temperature sensor might generate a digital signal periodically (say, once a second) indicating the current temperature. Alternately, the same sensor may send a signal only when the temperature deviates from its previous value (or a set norm) by certain amount [2,3].

In addition to the sensors that generate signals from physical observations, HUMS may also have virtual sensors (VS). A virtual sensor reads data generated by several sensors, processes the data, and generates high-level data useful for other application processes. Also, there may be several application processes running on the nodes that read the sensor values and compute factors describing the health of the structures.

SIMULATION MODEL

In order to analyze the performance of SCRAMNet in the structural health-monitoring environment, we have simulated the basic functionality of the system in terms of three major components---the links, the nodes (stations), and the data sources (sensors). We now describe how each of the components and their interactions are modeled.

The link

The link here represents a fiber optic, peer-to-peer connection between two adjacent nodes in a SCRAMNet. It has the following parameters: (i) Cable length, and (ii) Start- and end-node identifiers. Since the transmission speed and propagation speed are the same for all links, these parameters are associated with the network, and not specifically with a link.

During the period of message transmission, at the start-node, the link component is considered to be busy. The transmission time, of course, depends on the length of the message (or packet) and the speed of transmission.

To simulate transmission errors, each link has a "probability of error" (E) parameter to indicate the probability of a transmitted messages being received erroneously at the end-node due to transmission errors. When a node receives an erroneous message, it simply forwards it to its successor node. Finally, when the source node receives the erroneous message, it retransmits the original correct message.

The Node

The second component of the simulator is the node (or station). Nodes are connected in a ring topology with links. Each node contains three message queues: the transmit queue (T-Queue), the retransmit queue (RT-Queue), and the receive queue (R-Queue). Each of these queues is limited to 1k-byte of messages (approximately 68 of 15-byte messages) and work in a First-in-First-out (FIFO) fashion. The T-Queue stores messages generated by the applications at that station. These are the messages received by a node from the local host, but not yet transmitted. The RT-Queue is responsible for storing messages that are received from the previous node on the ring and not yet retransmitted. Finally, the R-Queue stores messages whose destination is the local host.

Each node is modeled to perform the following three basic functions. These are in line with the three data queues described above.

1. Receive messages from local data sources.
2. Receive messages from the network (from previous station).
3. Transmit received messages to the next node on the ring.

The RT-Queue messages have the priority over the T-Queue messages in transmission. However, if a new message arrives at the RT-Queue while another T-Queue message is being transmitted, the RT-Queue messages waits until the outgoing link is available. According to the register insertion protocol (of SCRAMNet), a node will certainly have a chance to place a new message on the ring when it receives its own previous messages from the network. Thus, the maximum delay that a node has to wait to place its local message on the ring, even under heavy load, is equal to the length of the ring. This gives each node a chance to place a new message on the ring at a maximum of a single ring length delay.

We now consider the error-handling aspect of a node. The error handling done at the node level is simple; a node keeps a copy of the last message sent in a buffer. When the node receives its message again from the network, it checks whether or not the message error bit has been set. If the error bit is indeed set, it simply retransmits the message. The message will be retransmitted even if the error has occurred after the destination correctly received it.

Data sources

In the simulation model, a data source (a sensor in HUMS) is modeled as a simple entity that generates messages periodically (with period "I") and sends them to its local node. A process at the node, after some initial processing (not modeled here), writes it into a memory location. The memory writes trigger a messages to SCRAMNet. The assumption of periodic message generation (as opposed to aperiodic messages) was simply to concentrate on other factors of study (e.g., number of nodes, distribution of load on the nodes, etc.). Each message is divided into a set of 4-byte packets. The generation period is varied between 1 and 60 microseconds.

Data filtering

SCRAMNet allows for data filtering to take place. When data-filtering option is used, a station does not transmit a value from a sensor unless it has changed from its earlier value. In other words, if the temperature in an engine part is constant over time, then even though the associated temperature sensor is generating the value of the temperature and sending it to its local node, the station does not transmit it since the value has changed. When this option is used, the data sources appear to be aperiodic for the network. These techniques remove much redundancy in cases of slowly changing data (such as temperature). This concept is supported in the simulator through a data filtering parameter (F) supplied to the data source. This parameter indicates the percentage of new data. In other words, F=0.0 indicates that all generated data is unique and that there is no redundancy (e.g., temperature is continuously changing). On the other hand, F=0.4 indicates that only 40% of the data is redundant or repeated. Thus, the traffic on the network due to this source will be reduced by 40% when data filtering option is used in SCRAMNet.

Simulation parameters

The simulator is provided with the following run-time parameters:

1. The number of data sources (D)
2. The number of stations (S)
3. The simulation time to run (in nanoseconds) (T)
4. The error factor (link errors) (E)
5. The filtering parameter (data filtering) (F)
6. The source interval (I)
7. The source distribution factor (SD)

Most of these parameters were discussed above. The simulation time determines the time to run a simulator and is expressed in microseconds. The last parameter, the source distribution factor, describes how the data sources are distributed among the nodes. When SD=1, the data sources are equally distributed among all the nodes. So all nodes have the same number of data sources. When SD=2, the data sources are distributed equally among alternate nodes along the ring. For example, if a node is assigned data sources, then neither of its neighbors has data sources. This parameter is used to study the effect of data distribution (keeping the same total load on the network) on the performance of the system.

RESULTS

In order to study the impact of implementing HUMS on a SCRAMNet, we conducted several simulations. In this section, we present a few of the results and summarize the overall results. In all the simulation runs, we assumed the following.

Equal spacing. Nodes are equally spaced along the network. We assume that the distance between successive nodes on the SCRAMNet ring network is equal. In particular, we assume that the distance is 100 meters. In addition, we assume the speed of propagation of the signal on the ring to be 200 km/millisecond (or 2/3 the speed of light). So the propagation delay is taken to be 0.5 microseconds along each link (between two successive nodes) of the ring network.

Fixed size messages. We assume that each message is 15-bytes long. Since SCRAMNet sends data as 4-byte fixed length packets, each message is sent as four data packets. In line with SCRAMNet's specification, we assume that it takes 613 nanoseconds (or 0.613 microseconds) to transmit a data packet at a node.

Message destination. To measure the message delay, we assume the node that precedes the source node on the network to be the destination. All results presented below assume the message delay to be "the interval between the instant a message arrives at a SCRAMNet node for transmission to the instant the message is correctly read by the station prior to the source node."

Network load. The network load is expressed in terms of total number of messages per microsecond generated by all the sources (cumulatively) on the network. This is indicated by LF or load factor.

Source distribution. For simplicity, we assumed the number of sources to be the same as the number of nodes on the network. However, we also experimented with the effect of load distribution on system performance by having different source distributions. The type of source distribution is indicated by SD. SD=1 indicates that the sources are equally distributed among all nodes. SD=2 indicates that every alternate node has two sources each.

Effect of system load on Message Delay

In order to determine the effect of total message load on the message delay, we have conducted several simulation experiments. However, in this paper we have included only a few key results. Figure 1 shows average delay under different network loads (in messages/microsecond). At low loads (LF <0.3), since the queueing delay is insignificant, the message delay is primarily dictated by the total length of the ring (in microseconds), the transmission time per message, and the number of nodes. In other words, it is independent of the actual LF factor. For loads higher than 0.3, but lower than 0.5, the delay increases somewhat linearly with LF. Beyond LF of 0.5, the delay starts increasing non-linearly. The delay characteristics for LF between 0.4 and 0.5 are shown in Figure 2. Under normal condition, the time a sensor generates data to the time it is received by all processes on the network seems to be quite reasonable, less than 150 microseconds. However, the behavior under peak loads needs further investigation. For example, if a problem has developed in one of the structures, it is likely that all sensors associated with that structure would start generating data. This may overload the network. In other situations where it is critical for the data to reach on time, there could be a delay. We are currently investigating the behavior of SCRAMNet under such "bursty" load conditions.

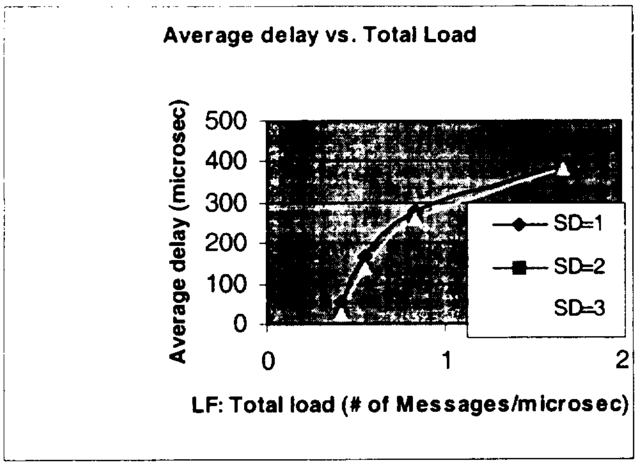


Figure 1. Effect of Load on Average Message Delay

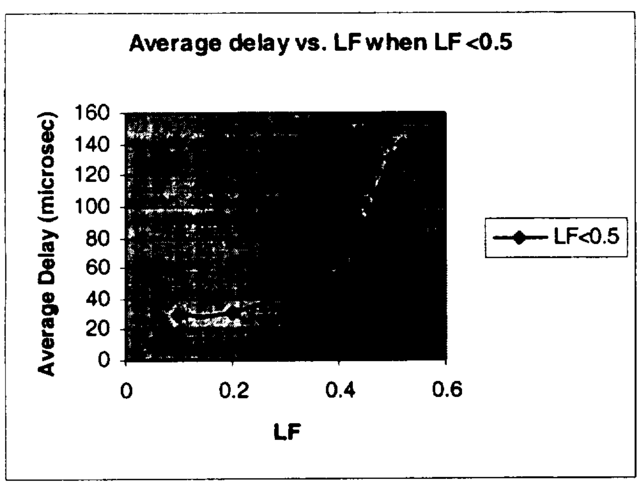


Figure 2. Average delay when LF < 0.5

Effect of Load on the Standard Deviation of Message Delay

As shown in Figure 3, the standard deviation is almost insignificant at LF less than 0.3. For loads up to 0.5, the standard deviation is more or less constant. For LF>0.5, the standard deviation increases---initially linearly and then exponentially. This is explained by the fact that the length of the transmission queue (T-Queues) and retransmission queues (RT-Queues) increases with the load. Since the length also varies significantly from node to node, the delay also varies. These are adequate for most HUMS applications. Once again, the behavior under heavy or synchronous loads need further study. Standard deviation of delay is also very critical in some HUMS. It basically determines the distribution of delays between the time a sensor generates data to the time its value is received by all the other processes or nodes in on the network.

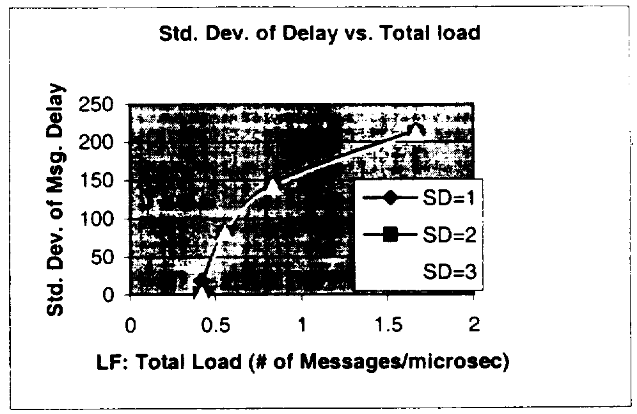


Figure 3. Effect of Load on Standard Deviation of Message Delay

Effect of Number of nodes

Figure 4 summarizes the effect of the number of nodes on the average message delay. As the number of nodes is increased, the total load also increases. At low loads (LF=0.1 and 0.2) the increase is linear which is mainly due to the increase in the ring length proportional to the number of nodes. For medium loads such as LF=0.4 also, the increase is somewhat linear with a higher slope. At high loads, the increase is expected to be exponential. The number of nodes is also a factor of great cost concern. Since adding an additional node to the SCRAMNet means adding an additional interface, it is also a cost consideration. In an aircraft scenario, where weight is a concern, we do not expect more than 20 to 30 nodes on a system. We expect that many sensors may be interconnected through some local multiplexing on to a single node. Thus, we do not expect this to be a signification limitation in HUMS.

Effect of Network Errors

In SCRAMNet, whenever a massage is found to be in error, the source detects the errors and retransmits the message. Thus, errors increase retransmissions and hence the load on the system. In addition, since we only measure the time the message was submitted to the time it was correctly received by the last node, retransmissions also increase the overall message delay. Figure 5 summarizes the effect of errors. At low loads, the effect is almost insignificant. Since SCRAMNet uses fiber-optic networks, we only considered low error rates. At high loads, however, the effect of errors is more prominent.

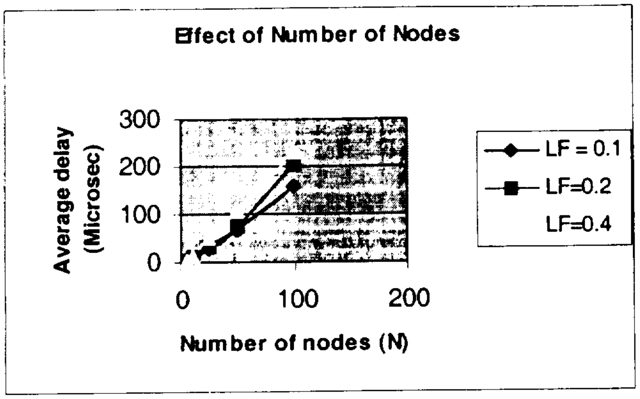


Figure 4. Effect of the Number of nodes on Average Message Delay (SD=1)

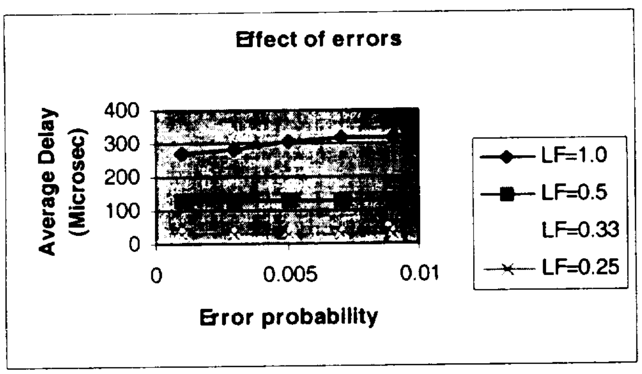


Figure 5. Effect of Errors on Average Message Delay (SD=1)

Effect of Data Filtering

As discussed in the background section, SCRAMNet has a feature of not sending data unless it is different from the earlier values. So when sensors report the same values continuously, these will not be sent to all nodes. Only when a reported (as written in its memory location) value has changed, the value is sent as a message. Figure 6 illustrates the effect of data filtering on average message delay. At low and medium load, the effect of data filtering is not apparent. However, at high loads, as the amount of filtered data (due to repetitive values) increases, the average delay decreases rapidly. Obviously, as the data filtering increases, the amount of real load on the system decreases, and hence a decrease in the average delay. Data filtering option is an extremely important option in HUMS where the legacy sensors periodically emit the same signals. In addition, if we were to use low-level data processing prior to placing data on the SCRAMNet, then the data filtering feature could be used by suppressing some minor changes not to be propagated on the network. We are still investigating different uses of this feature in HUMS.

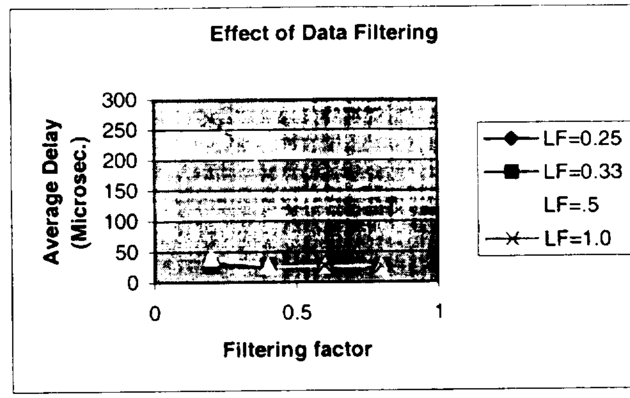


Figure 6. Effect of Data Filtering on Average Message Delay (SD=1)

Effect of adding an additional source/additional node

One of the frequent design decisions that a HUMS designer has to answer is: "If a new sensor is to be added to the HUMS system, should we connect it to an existing node or should we procure a new node, connect it to the SCRAMNet and then connect the sensor to the new node?" To answer this question, we made some preliminary runs. The results are shown in Table 1. The table illustrates four cases. For each case, the average delay is measured with

Case	Orig. Conf.	New SRC Added	New SRC + NEW node
I	99.1	139.8	167.8
II	106.5	126.8	143.4
III	121.6	131.6	141.0
IV	304.1	314.2	324.7

Table 1. Effect of additional sources/nodes

original configuration, when a new source is added to an existing node, and the new source added to a new node are measured.

In case I, with 5 nodes and 5 sources, with each source distributed to each node, the initial delay at LF=0.5 was 99.1 microseconds. When a new source (6th source) is added to one of the existing nodes, the delay increased to 139.8 microseconds, a 41% increase. However, when the new source was added to a new (6th) node, the delay was 167.8 microseconds, an increase of 69%. Similarly, in case II, with

10 nodes, adding the source to an existing node resulted in a 19% increase in delay, while adding a new node also resulted in 35% increase. The increase was much less in case III where 20-node system was considered. In case IV, with 10 nodes but an increased load factor (LF) of 1.0, the increase was much less significant. We are still investigating this behavior of the system. The effect of adding a new node or a new source needs further investigation. Some of the anomalies that we observed during the limited runs could not be explained. We are currently investigating this issue.

CONCLUSION

In the current work, we have modeled and simulated SCRAMNet for use as a backbone for a structural health and usage monitoring system (HUMS). One of the main concerns in a HUMS system is the ability of the system to deliver the sensor data to all processes on the network in a timely fashion. To this effect, we have represented the sensors as data generators and simulated the systems under various conditions. In particular, we observed the system behavior under different load conditions, with different number of nodes, different error rates, and data filtering rates. The current results indicate that at loads of 0.3 messages/microsecond or less, the system behavior is very stable. Even between loads of 0.3 and 0.5 messages/microsecond, the system behavior is predictable as the delay changes linearly. Beyond these loads, the system offers an average delays in the range of 350 microseconds and higher. The variance in the delay is also high at high loads.

As part of the future work, we plan to do a more rigorous study of the system. In particular, we plan to also take into account the effect of the processing load on the behavior of a node. The question of having heterogeneous data sources and their effect will also be answered. Another critical factor that needs to be studied in depth is the behavior of SCRAMNet when data bursts occur in HUMS. Whether the system can handle the generated loads or a prior pre-local processing at a structure needs to be done before putting the signal on the SCRAMNet is to be investigated further.

ACKNOWLEDGEMENTS

The work is sponsored in part by NASA Langley Research Center under grant NAG-1-2279. We wish to thank Mr. Rajendra Shirhatti, a graduate student in Computer Science at the Old Dominion University, for his effort in running several experiments using the simulator and tabulating the results.

REFERENCES

[1] Kent, R.M. and D. Murphy. 2000. "Health monitoring system technology assessments: Cost benefit analysis." *NASA Technical report.* NASA/CR-2000-209848.

[2] Mukkamala, R. 2000. "Distributed scalable architectures for health monitoring of aerospace structures." *Proceedings of IEEE DASC* (Philadelphia, PA, Oct. 7-13), 6.C.4.1-6.C.4.8.

[3] Mukkamala, R.; K. Bhoopalam; S. Dammalapati. 2001. "Design and analysis of a scalable kernel for health management of aerospace structures." *Proceedings of IEEE DASC* (Daytona Beach, FL, Oct. 14-18).

[4] "SCRAMNet+ Protocol Description." 2000. *Technical note 126.* Systran Corporation.

[5] "SCRAMNet Network: PCI Bus Hardware Reference." 1998. *Document No. D-T-MR-PCI-A-0-A6.* Systran Corporation.

[6] Moorthy, V.; M. Jacunski; M. Pillai; P. Ware; D. K. Panda; T. Page; P. Sadayappan; V. Nagarajan; J. Daniel. 1999. "Low Latency Message Passing on Workstation Clusters using SCRAMNet." *International Parallel Processing Symposium (IPPS'99).* 148-152.

[7] Jacunski, M.; V. Moorthy; P. Ware; M. Pillai; D. K. Panda; P. Sadayappan. 1999. "Low Latency Message-Passing for Reflective Memory Networks." *International Workshop on Communication, Architecture, and Applications for Network-Based Parallel Computing (CANPC '99).* pp. 211-224.

[8] Moorthy, V.; D. K. Panda; P. Sadayappan. 2000. "Fast Collective Communication Algorithms for Reflective Memory Network Clusters." *Fourth Int'l Workshop on Communication, Architecture, and Applications for Network-Based Parallel Computing (CANPC '00).* pp. 100-114.

[9] Hammond, J.L. and P. J.P. O'Reilly. 1988. *Performance analysis of local computer networks.* Addison-Wesley Publishing.

Specification and Evaluation of Software Architectures for Health Usage and Monitoring Systems (HUMS)[1]

R. Mukkamala[2] P. K. Anumula K. Bhoopalam M. Moharrum
Department of Computer Science
Old Dominion University
Norfolk, Virginia 23529-0162
USA
{mukka, anumu_p , kbhoopal, mohar_m}@cs.odu.edu

Abstract

Monitoring the health of aerospace structures is vital for aviation safety. In collaboration with NASA Langley Research Center, we have been investigating some of the key characteristics of software architectures for the health and usage monitoring systems (HUMS) for avionics. Due to the criticality as well as the cost of these systems, it is essential that candidate architectures (say from different vendors) be evaluated before a final architecture(s) is selected, developed, and implemented. In this paper, we describe a methodology to specify architectures and to evaluate them. Unlike most specification methods, we deal with the problem of specification and evaluation in an integrated way. So evaluation is not an after thought, but bears substantial influence on the specification method. We consider scalability, robustness, and flexibility as some of the architectural evaluation metrics to illustrate the methodology. We divide the architectural specification into three parts: functional, design, and performance. The paper describes the methodology and illustrates it with examples from HUMS.

Keywords: Architectural evaluation, architectural specification, avionics, distributed systems, flexibility, HUMS, performance evaluation, robustness, scalability, software architectures.

1. Introduction

Architectural decisions have a great impact on the quality of software systems. When acquiring a large software system that will have a long lifetime within the acquiring organization, it is important that the organization develop an understanding of the requirements for such architectures. This understanding allows an organization to more formally specify its requirements as well as evaluate the candidate architectures. A formal software architectural evaluation provides several benefits including:

- Allows the early detection of problems with the candidate architecture. It provides early insights into product capabilities and limitations.
- Allows for examination of the goodness-of-fit of the candidates to the functional needs as well as the performance requirements like reliability, scalability and maintainability of the desired systems.

---

[1] This work is supported in part by NASA Langley Research Center, Hampton, Virginia, under research grant NAG-1-2279.
[2] Contact author: E-mail: mukka@cs.odu.edu; Voice: (757) 683-3901; Fax: (757) 683-4900

- Formal specification and evaluation processes will force development organizations (vendors/bidders) to develop better and more suitable architectures (since they know the evaluation metrics ahead of time).

In Collaboration with NASA Langley Research center, we have been developing architectures for systems to monitor health and usage (HUMS) [10,11]. During this development and analysis process, we have realized that unless properly planned from the beginning, the task of evaluating the candidate architectures could become quite difficult or even impossible. Certainly, when we consider the complexity and criticality of systems like HUMS [8], and the large number of candidate architectures that can be expected from the bidders (due to the size and importance of the project), the evaluation problem could be quite difficult. This also means that there is a potential for a wide range of specification formats, nomenclature, different degrees of details about the candidate architecture, etc. Sometimes the data needed for evaluation may be hidden in hundreds of pages of architectural descriptions provided by the bidders. In order to make the task of the evaluator much more effective, we have arrived at a combined specification and evaluation methodology. In other words, the specification methodology takes into account the evaluation process and accordingly decides on the specification formats. Once the candidate architectures are submitted in the prescribed format, the evaluation process becomes rather straightforward.

In this paper, we present the proposed methodology and illustrate its usage with examples from HUMS. Clearly, this is a work-in-progress and we propose to improvise the methodology as more experience is gained in its usage. The paper is organized as follows. In section 2, we provide a background on some the earlier work in software architecture specification evaluation, and a few details on HUMS in avionics. In Section 3, we provide the HUMS reference architecture that is used as a basis for other illustrations in the paper. Section 4 deals with the specification methodology with examples from HUMS. In section 5, we describe the architectural evaluation methodology. Finally, section 6 summarizes our current work and describes our plans for the future.

2. Background

The work in software architecture specification [2,4,9] and evaluation [1,3,6] has been in progress for many years. In this section, we summarize a few contributions this area.

Hilliard et al. [6] highlight the importance of a standardized terminology for architectural description. The paper also provides insight into architecture evaluation using *iterators* and *discretes*. Iterators are similar to questionnaires as in [1]. Discretes are questions whose answers provide a simplified discrete scoring mechanism. This simplifies the aggregation and interpretation of results.

Lloyd and Galambos [9] discuss in detail the requirements of an "Architecture Description Language" to allow all professionals involved, to communicate and share concepts in a consistent manner. It emphasizes the importance of domain specific reference architectures as assets that can be brought to the table prior to system development. The paper also emphasizes the description of reference architectures at different levels of abstraction to promote reusability.

Abowd et al. [1] categorize architectural evaluation into two major categories: "Questioning" and "Measuring" techniques. Questioning techniques are further categorized into questionnaires, scenarios and checklists, each of which differ in their applicability. Measuring techniques include metrics that provide quantitative measures of the various performance characteristics of the architecture.

Kazman et al. [7] address the problem of architecture evaluation by using scenarios to gain information about a system's ability to meet desired quality attributes. This work addresses briefly the importance of common syntactical architectural notation that can be understood by all parties involved in the evaluation process.

Much of the work in architecture evaluation (e.g., [7]) deal with evaluating a specific architecture with respect to identifying critical modules of an architecture so that more effort may be expended to make the module more efficient. The evaluations are also used to identify modules that may be involved in too many activities (or scenarios) and hence need to be broken up into several modules. More importantly, the papers largely discuss architecture specification and evaluation as separate activities. In this paper, we take the view that specification and evaluation be dealt with in an integrated manner.

In regard to the application domain, the health usage and monitoring systems (HUMS), monitoring the health of certain aerospace structures has been shown to be a key step in reducing the lifecycle costs for structural maintenance and inspection [8]. Since the health of the structures ultimately determines the health of a vehicle, health monitoring is also an important prerequisite for improved aviation safety. The need for developing architectures for integrated structural health monitoring has been discussed in [10]. Some of the key characteristics of HUMS architectures are scalability, robustness, flexibility, and maintainability. Basically, in HUMS, the structural components of an aircraft are embedded with sensors. The sensors (periodically or aperiodically) send signals indicating the state of a structural component. The signals are received and processed by process controllers which in turn send them to higher levels. Since avionic structures are large, a distributed architecture is suggested to manage the sensors, the signals, and processes at all levels [10]. For further details, the reader may refer to [10,11].

3. HUMS Reference Architecture [10,11]

In this section, we describe the reference architecture that has been developed for HUMS [10,11]. The reference architecture is what the solicitor specifies in the request for proposals (RFP). We use this as a basis for specifying and evaluating the candidate architectures.

The proposed reference architecture is shown in Figure 1. It is compatible with the Generic Open Architecture (GOA) proposed as a standard for avionics systems [5]. It consists of six layers that may be logically divided into three parts. The lower part deals with sensors and low-level processing and control. The middle part deals with system level processing and maintenance. Finally, the upper part is related to the application software and interfacing with the user.

The sensor-layer (hardware) consists of signal emitters (sensors) that are representative of physical attributes such as temperature and pressure at a particular location (grid) on a structure. Some sensors may be emitting periodic signals while others may be emitting signals aperiodically. In addition, there could be heterogeneity in terms of the rates of emission, the types of signals, etc. The upper layers hide the heterogeneities from the application processes.

The low-level sensor processing and control layer (predominantly hardware) would have the ability to interact (turn-off, turn-on, and fetch signal) with the sensors. It would also include any analog-to-digital conversion needed for some of the sensors. It may include some simple fusion algorithms implemented in hardware or software. In essence, the lower part of the architecture offers services related to sensor data and control.

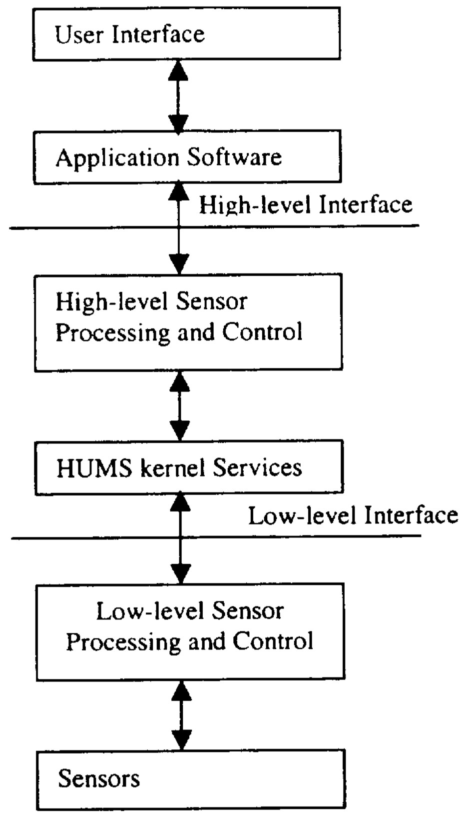


Figure 1. HUMS Reference Architecture

The HUMS kernel contains the infrastructure components to provide a robust, dynamic, and maintainable distributed sensor system [11].

The high-level sensor processing and control layer has the capability to control (via low-level sensor controllers) groups of sensors. In addition, it offers some complex fusion and integration routines. HUMS kernel would constantly monitor the processes in this layer to enhance the reliability of the system.

The application software layer consists of several installation or domain specific software needed by the end-user. Finally, the user interface offers interface to the end-user of the system. In case of HUMS, it could be the on-board staff such as the pilot or the ground staff such as engineers.

4. Specification Methodology

Keeping the architectural evaluation and its complexity in mind, we have divided the architectural specification into three categories.

(i) Functional specification
(ii) Design specification

(iii) Performance specification

The functional specification is mainly intended to specify the functionality of a proposed architecture. If an evaluator finds that a candidate architecture's functional specification is found to be incomplete with respect to the reference architecture, then the candidate architecture is immediately rejected. In other words, the functional specification acts as a necessary condition.

Once a candidate architecture meets the functional requirements, then the evaluator would look at the design specification. It gives details of the modularity of the candidate architecture. An evaluator uses it to check the functional coverage as well as organization and distribution of the modules.

Finally, the performance specification is used to specify the performance of the candidate. It has both quantitative and qualitative aspects. An evaluator looks at this in the final step.

4.1 Functional specification

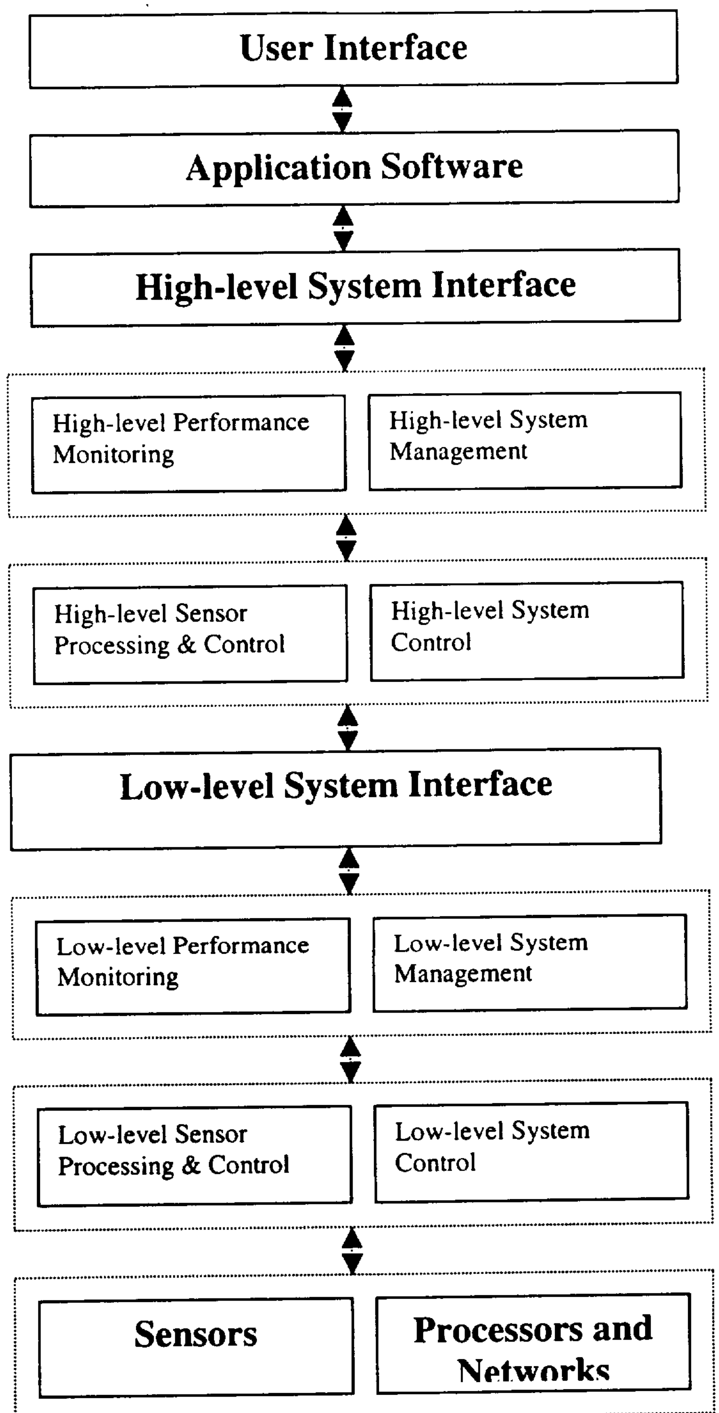
The main goal of a functional specification is to describe the functionalities supported by a software architecture. It describes, "What will be offered as a service" rather than "How it will be designed or implemented." Since our final goal is to evaluate several proposed candidate architectures, we use the functional specification as a first step in this process. In other words, by separating the functional specification from the design and performance specifications, we simplify the evaluation process. If a proposed architecture's functional specification does not meet the basic functionalities required, then it is immediately rejected thereby avoiding the unnecessary effort for further evaluation.

The HUMS reference architecture, as the name suggests, is only for reference. In other words, we do not expect all the proposed candidate architectures to be identical to the reference architecture. Some may propose architectures with more layers and some with less. In fact, some architectures may distribute the specified functionalities in ways quite different from that in the reference architecture. For this reason, we have designed the functional specification format so as to evaluate the completeness of an architecture in spite of apparent deviations.

The functional specification consists of two categories of information. First, descriptive information of the functionalities offered. Second, model coverage information in the form of tables.

Descriptive Information. The first component of the descriptive information is the layered architecture of the candidate. While the solicitor has provided the reference architecture (Figure 1), the bidder has provided the same for a candidate architecture (Figure 2). We use this as an illustration in this paper. For each layer in the candidate architecture, a brief description of its functionality is to be given. One of the reasons for using the term "functionality" rather than "function" or "procedure" is to avoid the confusion of the latter term when used in the context of design details or implementation. In other words, a functionality "A" specified in the architectural specification may actually be implemented using functions "F1, F2, ..." in the actual design and implementation. In this sense, functionality is at a much higher level than a function used in programming. This part of the specification would have a list of functionalities and their description for each layer.

Figure 2. Candidate Architecture



Consider the candidate architecture in Figure 2. It has nine layers with several layers consisting of two sub-layers. It also explicitly handles the infrastructure components such as processors and networks.

In addition to the layered architectural diagram, the functional specification should clearly describe the functionalities and interfaces offered by each layer. The interfaces could be described using any standard interface definition languages. Since this is only an architectural evaluation, and not a design and implementation stage, the definitions do not have to be very formal. Certainly, they would not represent the functional (or procedural) calls that they generally refer to at the design stage. Due to space limitations, we are omitting these details.

Coverage Information. The objective of this information is to ensure that the candidate architecture covers the minimum functionalities provided in the reference architecture. This information is captured in terms of two types of tables: Reference model coverage tables and Candidate model coverage tables.

A reference model coverage table is prepared for each layer in the reference architecture. The table lists all the functionalities specified by the reference architecture. For each listed functionality, an equivalent functionality (probably with a different name and at a different layer) in the candidate architecture will also be specified. If there is no equivalent, it will be left as blank. An example reference model coverage table for HUMS kernel layer is shown in Table 1.

Reference architecture layer: HUMS Kernel			
Reference Model Functionality	*Candidate Model Layer*	*Candidate Model Functionality*	*Description*
Dynamic System Management	Low-level System Management Layer	Life-cycle Management	Manages add/delete functions for components
	Low-level Performance Monitoring Layer	Monitoring Services	Monitor the performance of software and hardware components
Fault-tolerance	Low-level System Management Layer	Replica Management	Add/delete replicas
	Low-level Performance Monitoring Layer	Monitoring Services	Monitor the performance of software and hardware components
	Low-level System Management Layer	Life-cycle Management	Add/delete processes and sensors

Table 1. Reference Model Coverage Table: An example

A reference model coverage table describes which functional modules of the candidate architecture implement the functionalities of the reference architecture. It is quite possible that candidate architecture combines several functionalities of reference architecture into one module. It is also possible that candidate architecture implements additional functions not mentioned in the reference architecture. But these cannot be captured in this table but included in the candidate coverage table. For each entry in this table, a description of the interface is also specified.

Basically, this table enables an evaluator to ensure that the candidate architecture covers all the functionalities specified in the reference architecture.

The candidate model coverage table maps the candidate architecture functionalities to the reference architecture's functionalities. This table is necessary to account for those functionalities that have no equivalent functionality in the reference architecture. It is expected that for each such case, there would be an explanation indicating the reason for such additions. Table 2 shows part of a candidate model coverage table for the candidate architecture in Figure 2. It may be noticed that the candidate architecture has a functionality not included in the HUMS reference architecture. In addition, the architecture appears to be more modular in terms of functionality than the reference architecture.

Candidate Architecture layer: Low-level Performance Monitoring			
Candidate Model Functionality	*Reference Model Layer*	*Reference Model Functionality*	*Description*
Monitor sensor components	HUMS Kernel	Monitoring and relocation	Monitors sensor component's health
Recording component failures	HUMS Kernel	Monitoring and relocation	Record component failures
Reporting component failures	None	None	Reports failures

Table 2. Candidate Model Coverage Table: An example

In addition to the above coverage tables, the methodology suggests the inclusion of high-level data-flow and control-flow diagrams to illustrate the interactions among the layers. Due to space limitations, we have not included example data-flow and control-flow diagrams here.

4.2 Design specification

While the functional specification is intended to describe an overview of the candidate architecture, the design specification deals with the details of the modules and interactions between them that make the promised functionality a reality when the architecture is implemented.

In this paper, we assume that a functionality described in the functional specification may be implemented using one or more interacting modules. A module is a logical entity and may consist of a several procedures and functions determined at the time of implementation. But that level of detail is not available at the time of architectural specification and evaluation. So, in this paper, the lowest level we refer to is a module.

As mentioned before, one of the challenges that an evaluator faces in evaluating several architectures from different vendors/bidders is the variety in the terminology, the layers, the functionalities, and the modules. Our method simplifies this process by including the following details for the candidate architecture expressed in terms of functionality in the reference architecture. While the functional specification dealt with the functionality, the design specification goes further into the module level.

1. **Notation/terminology.** Here, the vendor/bidder needs to provide the names of modules that implement the named functionality. While the solicitor provides a list of sub-functionalities needed to achieve certain functionality, the bidder needs to name the modules in the candidate architecture that implement that functionality. This is necessary to compare architectures that may use different terminology for the same module. In case, the candidate architecture implements additional functionality beyond the pre-specified list, then provision is made to specify those modules.

   As an example, consider the *data-storage* functionality of the reference architecture. The solicitor has identified the following (Table 3) sub-functionalities needed for buffer-storage. The bidder/vendor has then identified the names of modules (shown in italics) in the candidate architecture that achieve the specified functionality. In Table 3, the solicitor specified four sub-functionalities. The bidder has identified the names of the modules in the candidate architecture that cover them. In addition, the candidate architecture provides three more sub-functionalities not specified in the reference architecture.

Functionality: Data-storage	
Sub-functionality (Provided by solicitor)	**Name of the module** (provided by bidder)
1. Interfacing with sensor controllers	*Storage-interface*
2. Managing temporary data	*Buffer manager*
3. Managing permanent data	*File manager*
4. Receiving data into buffers	*Buffer manager*
5. Any other sub-functionalities	*(i)* *Buffer-overflow manager* *(ii)* *Buffer reallocation manager* *(iii)* *Buffer error manager*

Table 3. Design specification---Notation/terminology: An example

2. **Macro organization.** This describes how the components specified in (1) above are organized and how they interact. For each module, the bidder needs to mention which of the following types of organization is used:
   - Centralized organization (Only one module for the entire system)
   - Fully autonomous organization (Several modules independently operating at different parts of the system)
   - Hierarchical (Modules of the component are distributed but they have a hierarchical control structure
   - Fully distributed (Modules of a component are distributed but the distribution is transparent to the user)
   - Other organizations

   For example, Table 4 illustrates an example macro organization for the data-storage functionality of Table 3.

3. **Functionality distribution.** In addition to the organizational information provided in (2) above, the bidder is expected to specify the distribution of the modules in the overall system. Figure 4 has a sample distribution. While it does not specify the implementation details, it does provide considerable information to the evaluator about the structure of the system. For example, it appears from the given diagram that the architecture consists of clusters of nodes, each node providing either a distinct service or a replicated service.

While the figure does not indicate whether LFC1 and LFC4 or replicas, it does indicate that the life-cycle service (LFC) in the system will not be completely be affected by a single node failure in the system. Whether the functionality provided by LFC1 is replicated elsewhere in the system is to be determined only from the performance specification. For complex systems, it is best to spread the functionalities among several distribution diagrams. That should be left to the discretion of the bidder.

4. **Interactions among modules.** In order to evaluate the modular nature of the design, the specification should include charts describing the interactions among the modules in implementing a specific candidate functionality. Similarly, a chart describing the interactions among modules corresponding to different functionalities at the same layer is also required. (Due to limited space, we are not including examples here).

Name of the module (provided by bidder)	**Macro -organization**
Storage-interface	
Buffer manager	*Distributed*
File manager	*Centralized*
Buffer manager	*Distributed*
Buffer-overflow manager	*Distributed*
Buffer reallocation manager	*Centralized*
Buffer error manager	*Hierarchical*

Table 4. Design specification---Macro organization: An example

4.3 Performance Specification

This is an important and often neglected part of architectural specification. But since our method integrates specification and evaluation steps, the performance specification takes an important role.

The primary performance concerns in HUMS are: scalability, robustness, and flexibility. Hence, the architectural specification evaluation will concentrate on these three measures. Since the solicitor knows the primary issues related to each of the measures with respect to his particular system, we require him to formulate the requirements in terms of questions and scenarios. For each functionality specified in the functional specification, the solicitor presents questions and scenarios at the time of solicitation (RFP). The questions and scenarios also help the evaluator(s) in understanding a proposed architecture to the desired abstraction level. The process forces the solicitor to formulate his requirements in a more precise manner. The technique is also useful to the bidder since he knows exactly what the solicitor wants and does not have to guess using the often-ambiguous requirement documents. We find this technique to be much more useful when compared to the traditional way where the performance details are hidden in a huge descriptive text.

To illustrate the efficacy of this method, let us consider the buffering functionality in the reference architecture. This is part of the HUMS kernel layer. It receives data from the sensor controllers (via low-level sensor processing) and makes it available to other high-level processes [11]. So it has the data storage and data retrieval as sub-functionalities. In this paper, we consider only the data storage aspects for illustration. We refer to it as a data storage module.

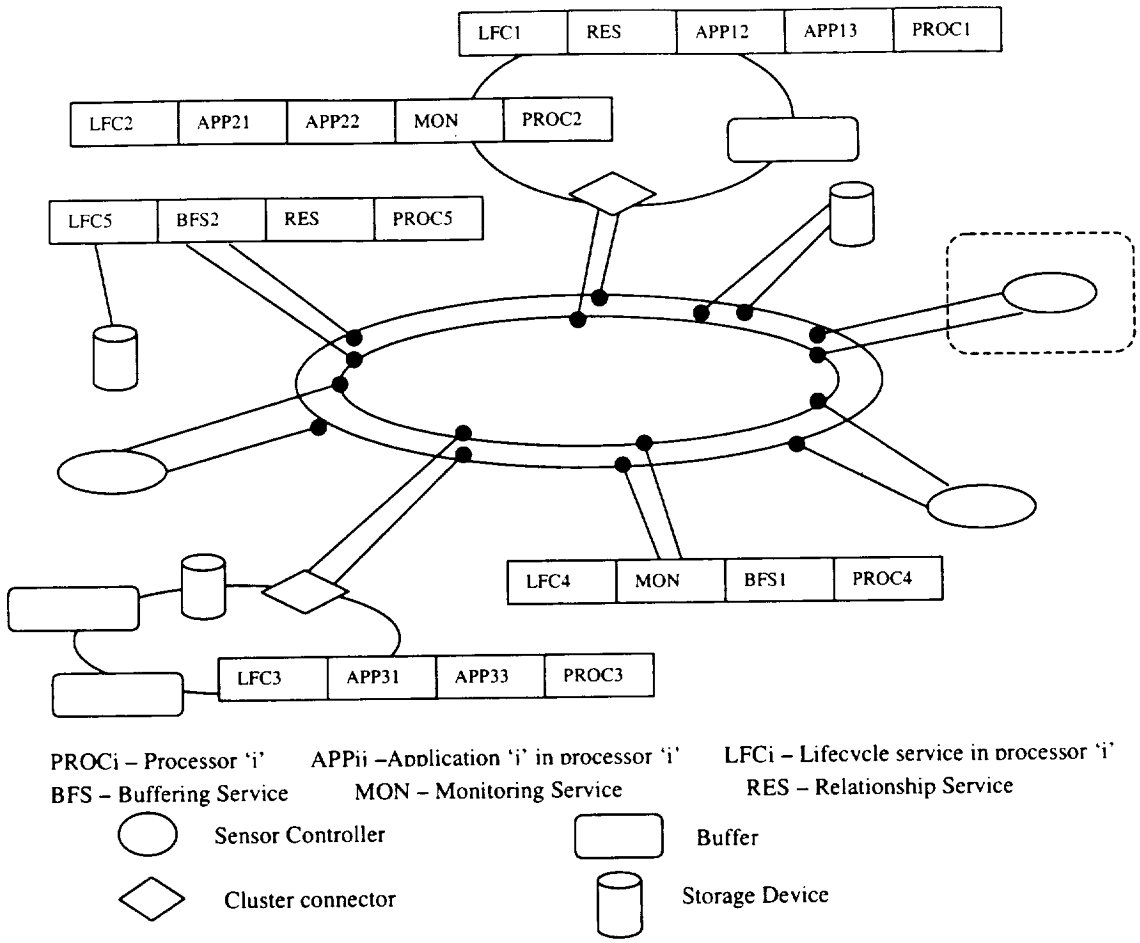


Figure 4. Module Distribution for Life-cycle Service (LFC): An example functional distribution

In terms of scalability, HUMS is mainly concerned about the ability to add more sensors, to be able to handle higher data rates, and to improve the response times. So these concerns are formulated as part of the design specification in Table 5. Due to space limitation, we have included only a few of the concerns. The main objective here is to identify the architectural limitations in terms of different scalability issues.

Similarly, the robustness concerns are illustrated in Table 6. Here, the idea is to identify what types of failures the architecture can tolerate and what it cannot tolerate. This helps the evaluators in identifying whether or not failure-handling mechanisms are incorporated in candidate architecture. Some of the flexibility concerns are included in Table 7.

While most questions in Tables5-7 are phrased as questions with Yes/No responses, some kind of explanation with the responses is implicit. In addition, other qualitative and/or qualitative questions may be included as part of the performance specification.

Tables of *selected scenarios* are also to be included for the metrics of concern. For example, in the case of robustness, one may want to know the consequence of a combination of failures. This will not be so evident from the simple questions in Table 6. Due to space limitation, we have not included such tables here.

5. Architectural Evaluation

Once we have the specification of the candidate architectures, the next step is to evaluate the architectures. Since the specification methodology was developed keeping this step in mind, evaluation should be fairly straightforward.

As described in section 4, the specification consists of three parts: functional, design, and performance. So each of these parts need to be carefully evaluated. It is important to evaluate these three in the respective order since it also indicates the order of importance. An architecture that does not meet the functional requirement may be rejected immediately even when its design and performance are noteworthy. Similarly, candidate architecture with good functionality and design may be rejected due to unacceptable performance. We now give a brief description of the evaluation process.

5.1 Evaluation of functional specification

Since the main objective of the functional specification is to express the basic functionality offered by candidate architectures, it should be evaluated in terms of compatibility and completeness of the offered functionality with respect to the desired (reference) functionality.

Clearly, the layered architecture should provide a broad guideline about the modularity and flexibility of the overall architecture. Sometimes too many layers may mean significant overhead and probably performance deficient systems [12]. In addition, the description about the functionality offered by each layer confirms to the evaluator whether the candidate is to be evaluated further. The description of the interfaces of each layer helps in this part of the evaluation.

The coverage information is probably a more formal resource to verify the completeness of the candidate architecture. Using the reference architecture coverage tables, an evaluator can clearly decide whether or not the candidate meets the basic functional requirements. In addition, the candidate coverage tables indicate what additional features the candidate architecture offers. This is very helpful since often the solicitor may have missed some characteristics that are captured by the expert bidders.

At the end of this stage of evaluation, the evaluator should be able to have a first-cut at the candidates. If they don't meet the functional requirements, they are rejected. Otherwise, they go on to the next stage of evaluation.

5.2 Evaluation of design specification

In this step, the evaluator looks at the some of the design details of the architecture. The main concerns here are the modularity, organization, and distribution. The modularity of the architecture may be evaluated based on how the functionality is subdivided further and how each subfunctionality is divided into modules. While the design should certainly include the subfunctionalities required by the bidder, the additional sub-functionalities implemented also helps the evaluator rate one architecture as superior to the other. For example, in Table 3, the candidate architecture included three additional functionalities for error handling not specified in the original reference architecture.

The organization and distribution of the modules is evident through the macro organization and distribution charts. For example, a centralized module is more prone to failures than a distributed module. However, a distributed module may have high performance overhead in terms of communication and synchronization cost. While the actual impact may not be so evident just from the design specification, an experienced evaluator can make a judgment or look for more evidence through the performance specification.

The distribution diagram gives a pictorial view of the distribution of the functionality and modules. For example, from Figure 4, it is evident that the life-cycle services is distributed all along the system and not just concentrated at one place. This could be a distributed and autonomous organization. However, if one service component is acting as a backup for the other, then some robustness could be achieved. But these facts are not so evident from here. Only the potential can be evaluated from here. The information in the performance specification could either confirm this hypothesis or refute it.

In summary, the design specification further helps the evaluator to make conclusions about the candidate architectures. But much is to be gained from the next stage---the performance specification evaluation.

5.3 Evaluation of performance specification

Much of the performance-related questions or concerns are to be answered at this stage. Of course, it is assumed that the solicitor has posed all relevant questions in a way for the evaluator to rank the architectures.

The information provided in performance specification may be both quantitative and qualitative. Of course, we expect most quantitative information to be in the form of ranges rather than absolute point estimates (since this is still at the architectural stage). With each response, especially when warranted, the bidder is expected to provide an explanation or justification for the answer provided. These would help the evaluator in assessing the architectures in a specific perspective. The verbal explanation help the evaluator develop a better picture of the architecture.

Depending on the importance of different factors (e.g., low communication versus low processing overhead), the evaluator may have been provided with some weights for different metrics. This enables the evaluator to arrive at one or more metrics representing the goodness or the suitability of candidate architecture to the needs of the solicitor. We expect the metrics to be based on different measures of interest such as sensor scalability, processing scalability, robustness to node failures, etc. Due to limited space, we do not include the details here.

Similarly, the manner in which a candidate architectures addresses each scenario also gives insights to an evaluator with respect to its design and performance. (Due to space limitation, we do not discuss it further here.)

By the end of this step, we assume that the evaluator is able to present both qualitative and quantitative evaluation of candidate architectures to the solicitor. We expect this step to cut down the number of candidate architectures, so the selected ones with potential could be asked to submit the next level of design and performance details or build prototypes.

6. Summary and Future work

In this paper, we proposed a methodology to specify and evaluate architectures. The main feature of our methodology is that both specification and evaluation are dealt with in an integrated manner. We divided the specification (and evaluation) into three parts: functional, design, and performance. Each part has more detailed information than the earlier parts. In addition, by using this step-wise approach, the evaluator is able to eliminate inappropriate candidate architectures at an early stage itself.

While the methodology is developed in the context of architectures for health usage and monitoring systems for avionics, we expect it to be equally applicable for other architectures. At this stage, the methodology has been discussed with the project sponsors (NASA Langley). So far it has been applied on some parts of the HUMS. In future, we plan to extend it to the entire system so that the sponsors may use it in their solicitation and evaluation processes. As we gain more experience in using the methodology, we plan to revise it and make it more comprehensive and usable.

References

[1] Abowd, G., L. Bass, P. Clements, R. Kazman, L. Northrop, and A. Zaremski, "Recommended Best Industrial Practice for Software Architecture Evaluation," Technical Report, Software Engineering Institute, Carnegie Mellon University, *CMU/SEI-96-TR-025*, Pittsburgh, PA, 1996.

[2] Bass, L., P. Clements, and R. Kazman, Software Architecture in Practice, Addison—Wesley, 1997.

[3] Briand,, L., J. Carriere, R. Kazman, J. Wuest, "COMPARE: A Comprehensive Framework for Architecture Evaluation," *European Conference on Object-Oriented Programming (ECOOP), Workshop on Software Architectures*, 1998, Brussels, Belgium.

[4] Clements, P., "A Survey of Architectural Description Languages," Proc. *8th International Workshop on Software Specification and Design*, Paderborn, Germany, 1996.

[5] *Generic Open Architecture (GOA) Framework*, Society of Automotive Engineers Inc., SAE AS48983, 1996.

[6] Hilliard, R., M. J. Kurland, and S. D. Litvintchouk, "MITRE's Architecture Quality Assessment," *1997 MITRE Software Engineering and Economics Conference*, 1997.

[7] Kazman, R., G. Abowd, L. Bass, and P. Clements., "Scenario-Based Analysis of Software Architecture," *IEEE Software*, Vol. 13, No. 6, November 1996, pp. 47-56.

[8] Kent, R. M. and D. Murphy, 2000, *Health monitoring system technology assessments: Cost benefit analysis*, NASA Technical report. NASA/CR-2000-209848.

[9] Lloyd, P.T.L., and G. M. Galambos, "Technical Reference Architectures," *IBM Systems Journal*, Vol. 38, No. 1, 51-75, 1999.

[10] Mukkamala, R., "Distributed scalable architectures for health monitoring of aerospace structures," *19th Digital Avionics Systems Conference (DASC'00)*, Philadelphia, PA, October 7-11, pp. 6.C.4.1-6.C.4.8, 2000.

[11] Mukkamala, R., K. Bhoopalam, and S. Dammalapati, "Design and analysis of a scalable kernel for health management of aerospace structures," *20th Digital Avionics Systems Conference (DASC'01)*, Daytona Beach, FL, October 15-19, pp. 3.D.2.1-3.D.2.10, 2001.

[12] Tanenbaum, A.S., Computer Networks, Third Edition, Prentice Hall PTR, NJ, 1996.

Layer:	HUMS Kernel
Functionality:	Buffer-Service
Sub-functionality:	Data Storage
Performance Measure:	Scalability
Design-related questions:	
(i)	What is the limit (if any) on the number of sensor controllers a data storage module can handle?
(ii)	If there is a limit, which resource(s) is the bottleneck?
(iii)	Can the limit be extended when the bottleneck resources are increased (in number or speed)?
(iv)	What is the limit (if any) on the data rate from sensor controllers a data storage module can handle?
(v)	Is the data rate limited over the cumulative rate (over all sensor controllers) or per sensor controller?
(vi)	If there is a limit, which resource(s) is the bottleneck?
(vii)	Can the limit be extended when the bottleneck resources are enhanced (in number or speed)?
(viii)	Can the performance (response time, etc) of the module be improved by increase in the number of buffers, the number of processors and/or their speed, the number of data servers (those that manage the buffers), or the number of file servers (those that save the data for later retrieval) is increased?
(ix)	What other additional resources can improve the response time?
(x)	Can the number of data-storage modules be increased to enhance the data rates, the number of sensor controllers handled, and the response time?

Table 5. Performance Specification---Scalability Issues

Layer:	HUMS Kernel
Functionality:	Buffer-Service
Sub-functionality:	Data Storage
Performance Measure:	Robustness

Design-related questions:

(i) What is the impact (if any) of the failure of a sensor controller (process), a buffer controller, a data server, or a file server on the data-storage module?

(ii) What is the impact (if any) of corruption of received data, data in buffers, data in file servers?

(iii) What is the impact (if any) due to overload (i.e., is there a possibility of losing data if the module can't attend to the incoming data immediately)?

(iv) What is the impact (if any) when buffers are full or the secondary storage is full?

(v) What is the impact (if any) of the unavailability (node failure/communication failure) of the node(s) executing the buffer controller, the data server, or the file server?

(vi) Is there a mechanism to handle loss of data between sensor controller and this module (i.e., sensor controller sent the data, but it never reached this module)?

(vii) Is there a mechanism to handle loss of data between buffer controller and data server

Table 6. Performance Specification---Robustness Issues

Layer:	HUMS Kernel
Functionality:	Buffer-Service
Sub-functionality:	Data Storage
Performance Measure:	Flexibility

Design-related questions:

(i) Is it possible to replace/modify the following without changing the rest of the system? If there is an impact, state the impact or changes that need to be carried out and the extent of the changes.
- (a) Sensor controller output data format
- (b) Type of sensor controllers (e.g., unintelligent to intelligent or vice versa)
- (c) Buffer controller, data server, or file server (e.g., a modified and improved version is now available)

(ii) Is it possible to carry out the following changes without impacting the rest of the system? If there is an impact, state the impact or changes that need to be carried out and the extent of the changes.
- (a) Increase/decrease in the number of data storage modules.
- (b) Increase/decrease in the number of sensor controllers sending data to a module
- (c) Increase/decrease in the number of data servers and/or file servers
- (d) Increase/decrease in the number of buffers
- (e) Increase/decrease in sensor controller output data
- (f) Increase/decrease in the load at the node(s) where the buffer controller/data server/file server are executing
- (g) Increase/decrease in the processing speed of the node(s) where the buffer controller/data-server/file-server are executing
- (h) Reallocation of sensor controller output among data storage modules
- (i) Other reallocations of data at data servers and file servers

Table 7. Performance Specification---Flexibility Issues

MS Project Presentation

Design and Evaluation
of
Structural Health and Usage Management System

Project Funded by NASA Langley Research Center

Advisor: Dr Ravi Mukkamala

Srihari Dammalapati

Kailash P. Bhoopalam

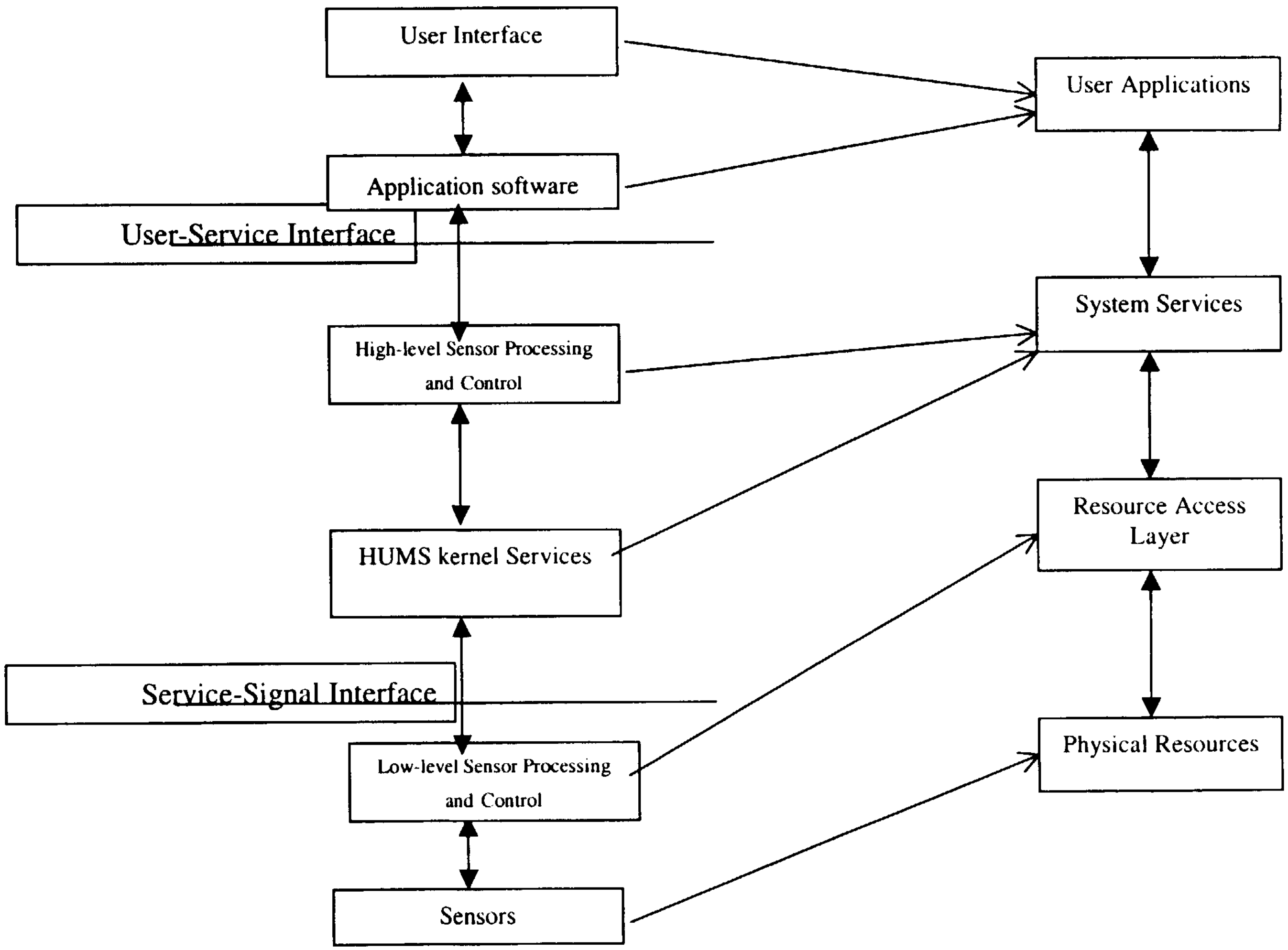
Introduction

- Structural Health, What is it?
- Why is a management system needed?
- Work that has been going on in the development of such systems.
- Work that we have done over the past year.

Contents of the Presentation

- Functional Specification of the HUMS reference model
- Design and Implementation of the HUMS kernel
  - » Srihari
- Reliability, scalability and flexibility enhancement mechanisms in the Layered Model
- Architectural Proposals and Evaluation methods
  - » Kailash

HUMS Reference Model



Figure

Functional Specification

- Sensors
  - Primary data producers.
  - Can be analog or digital.
  - Emit signals that can be consumer ready or require further processing.
  - Could emit in periodic intervals or only during perceived hazardous conditions.
  - Intelligent or dumb.

Functional Specification, contd.

- Low-Level Processing Layer
  - Analog to Digital Conversion
  - Signal Transformation
  - Simple Sensor fusion (Usually competitive)
  - Simple error control
- Sensor fusion
  - Complimentary
  - Competitive
  - Collaborative

Functional Specification, contd.

- HUMS Kernel has the following services
  - Data Buffering
  - Maintaining Structural component and HUMS component relationship
  - Lifecycle services for managing component lifecycle
    - (addition, removal and updating of components)
  - Monitoring and relocation services to enhance system reliability
    - (Monitoring for failure, non-optimal performance of HUMS components)

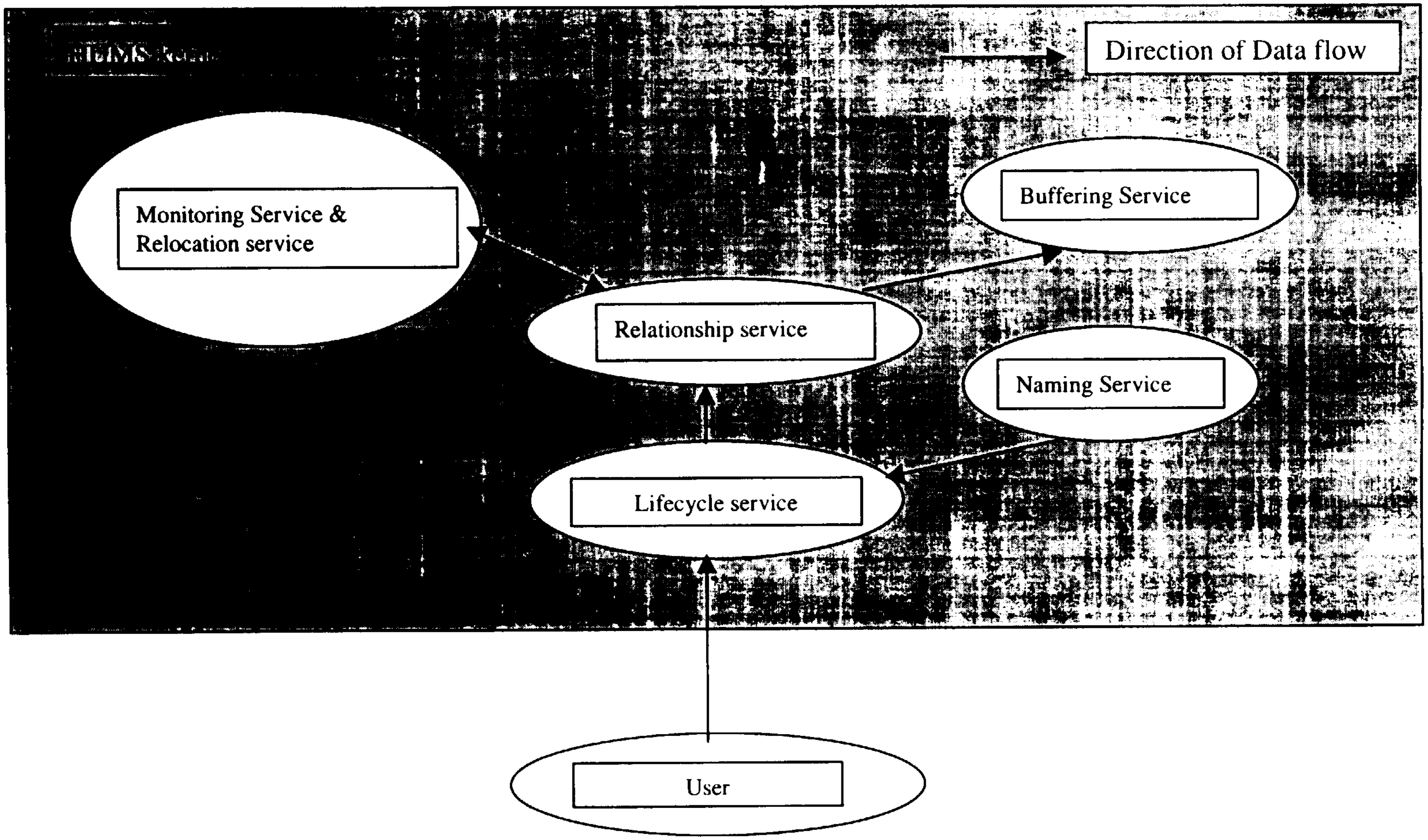
Functional Specification, contd.

- High-Level Processing and Sensor Control
  - Complex sensor fusion processes.
  - Analyze large quantities of data and store the results of their analysis.
- Application Processes
- User Interfaces
  - They could be domain-specific systems that perform further analysis of processed data.
  - These processes can have user-interfaces of their own or there can be a generic user interface.

The HUMS Kernel

- The objectives of the HUMS kernel are provided as services to the HUMS systems
  - Reliability (Reduce process failures)
  - Robustness (Allow processes to react gracefully during component failure)
  - Provide Services for system maintenance (addition and deletion of components)
  - Provide services for access to sensor data

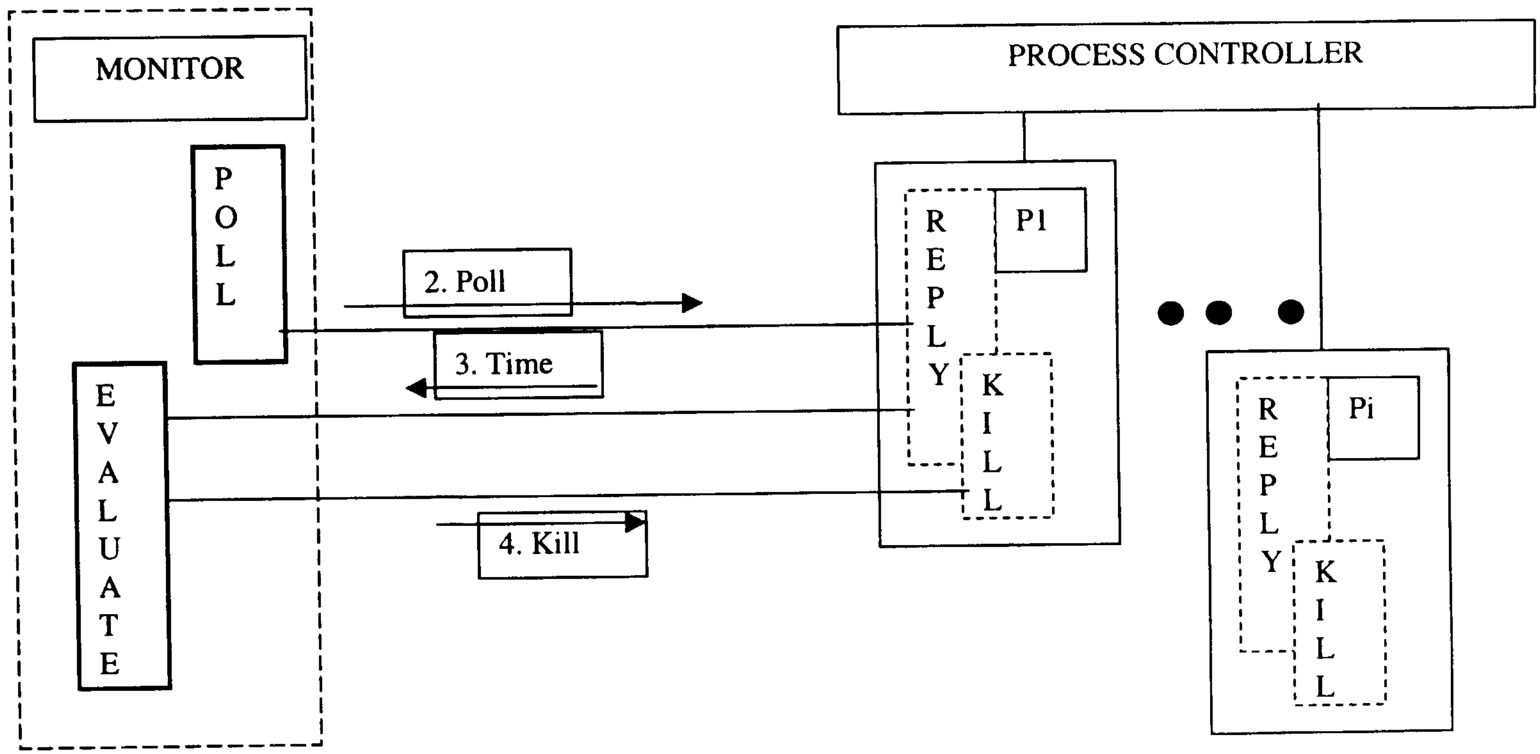
Kernel Service Interactions



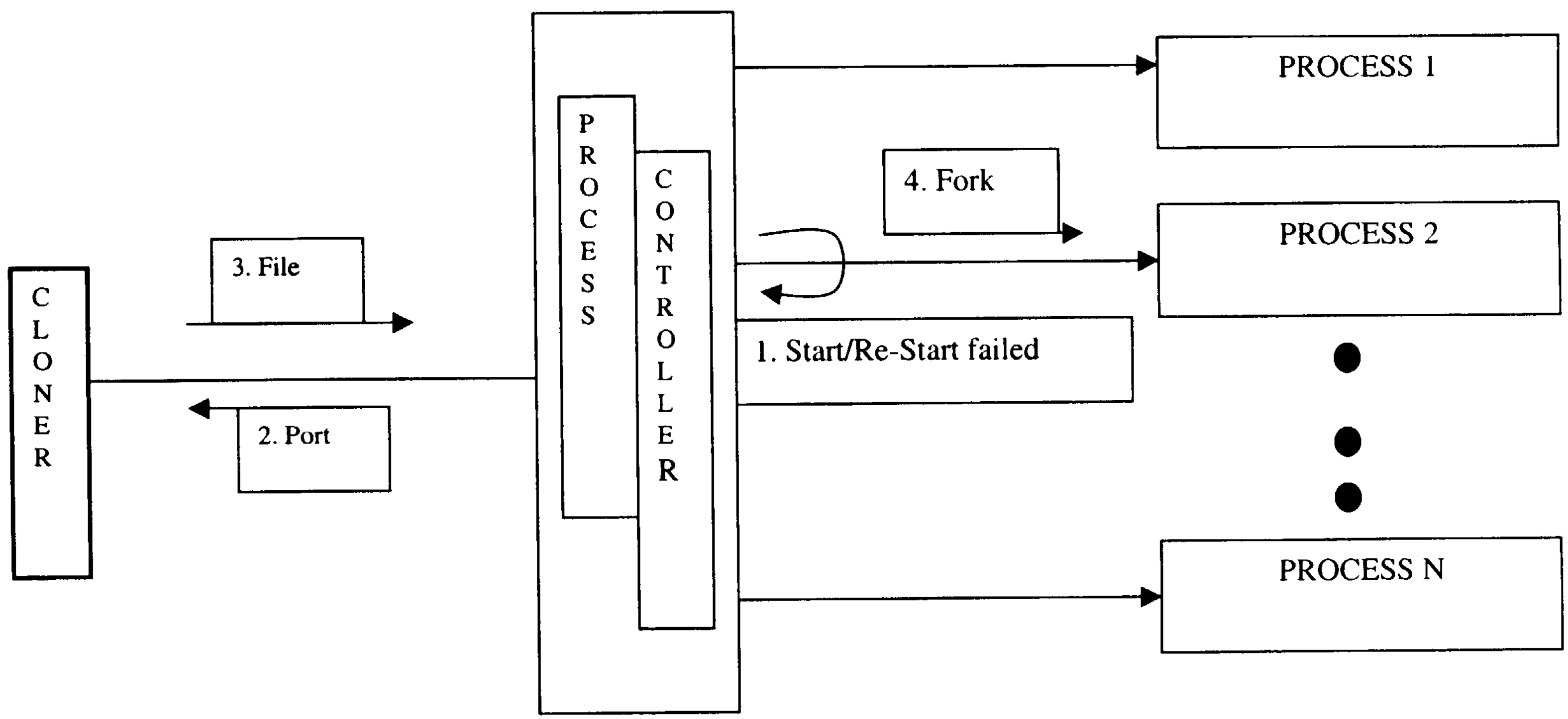
Monitoring and Relocation Services

- Purpose
  - Increase the reliability of the services in the system.
- The Monitoring and Relocation services consist of the following functionalities
  - Polling
  - Process Evaluation
  - Cloning

Polling and Evaluation



Cloning

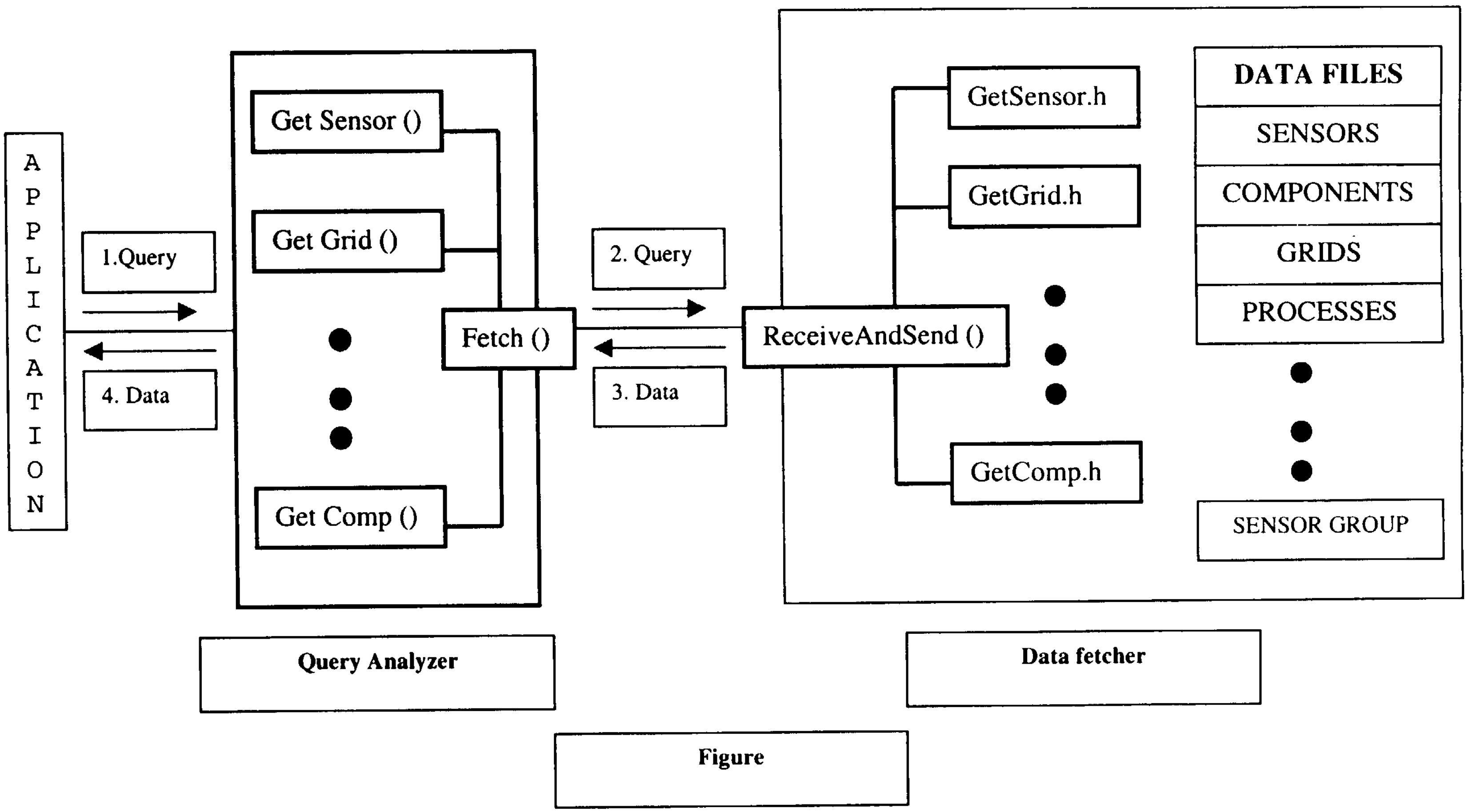


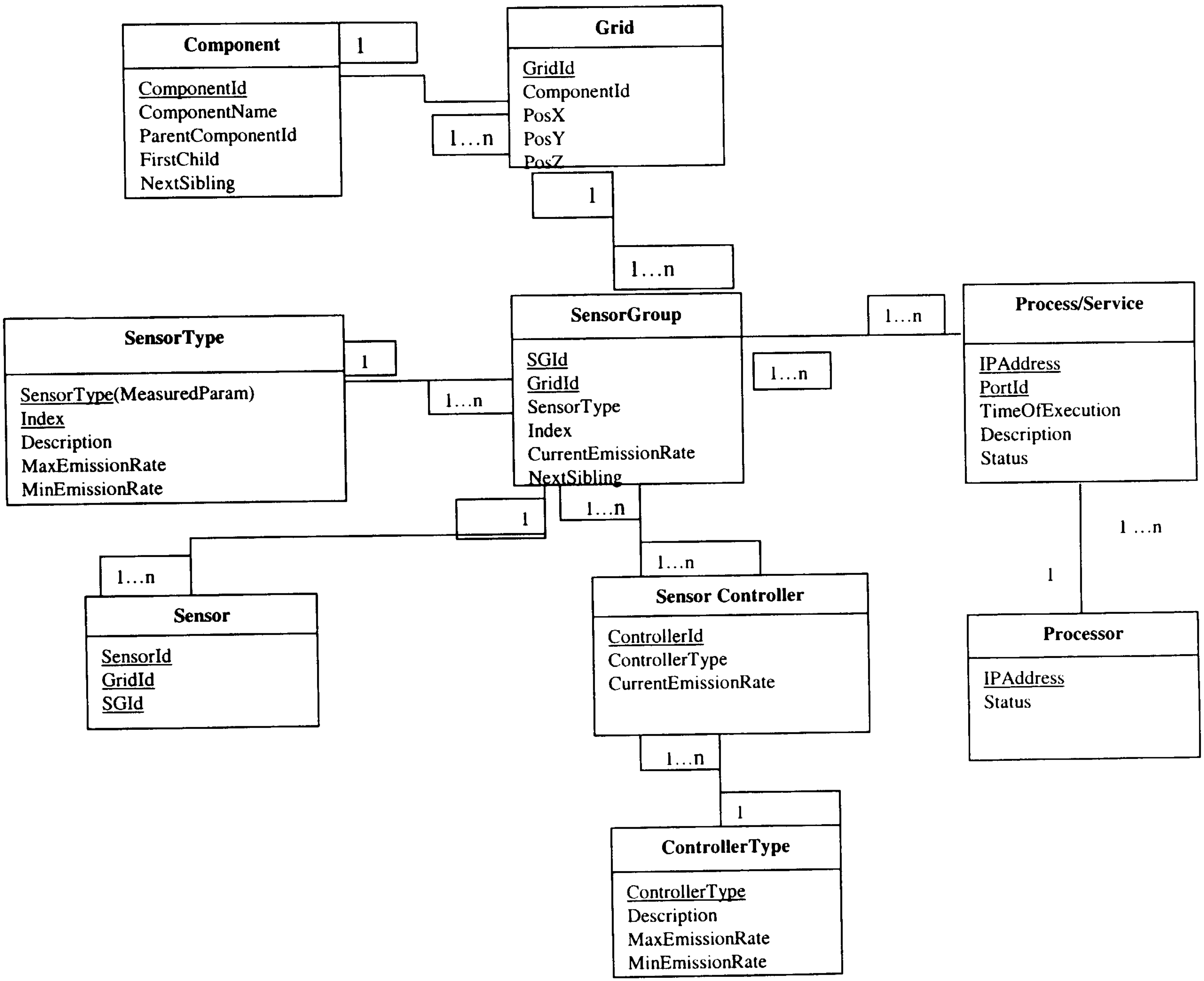
CLONER
3. File
2. Port
PROCESS
CONTROLLER
1. Start/Re-Start failed
4. Fork
PROCESS 1
PROCESS 2
PROCESS N

Relationship Service

- Maintains a hierarchy of Structural components of an aircraft.
- Maintains a relationship of HUMS components, the relationships can be broadly classified into the following
  - <Component, Grid>
  - <Grid, Sensor>
  - <Processor, Process>
- Mechanisms for storing and retrieving data.

Data Retrieval Mechanism





Component
ComponentId
ComponentName
ParentComponentId
FirstChild
NextSibling
1
1...n
Grid
GridId
ComponentId
PosX
PosY
PosZ
1
1...n
SensorType
SensorType(MeasuredParam)
Index
Description
MaxEmissionRate
MinEmissionRate
1
1...n
SensorGroup
SGId
GridId
SensorType
Index
CurrentEmissionRate
NextSibling
1...n
1...n
Process/Service
IPAddress
PortId
TimeOfExecution
Description
Status
1
1...n
1...n
1...n
Sensor
SensorId
GridId
SGId
Sensor Controller
ControllerId
ControllerType
CurrentEmissionRate
1 ...n
1
Processor
IPAddress
Status
1...n
1
ControllerType
ControllerType
Description
MaxEmissionRate
MinEmissionRate

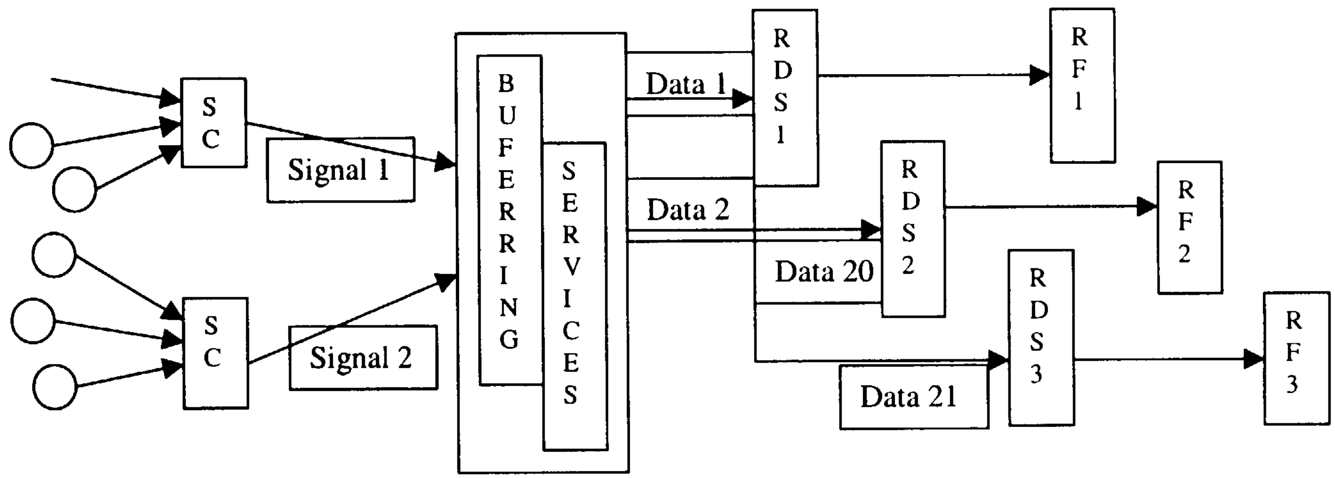
- Component
  - Provides information about the structural component of the aircraft and allows component hierarchy to be maintained.
- Grid
  - A Basic unit of interest in an aircraft that exists at the leaves of the structural component hierarchy.
- Sensor Group
  - A collection of sensors that measure the same physical parameter type (temperature, pressure, etc) *in a grid*.
- Sensor Type
  - Sensors that measure the same physical parameter.

- Sensor
  - Identifies uniquely every sensor.
- Sensor Controller
  - Refers to low-level sensor-controllers that process signals from the sensors.
- Controller Type
  - Refers to controllers that handle a particular parameter type of sensors signals.
- Processes
  - Application processes or High-level sensor controllers that perform complex processing sensor signals.
- Processor
  - Processors in which processes reside.

Buffering Services

- The buffering services provide for sensor data to be stored so that they can be accessed under the following conditions:
  - When a Process's requirement for data is not in synch with the emission of data by the sensors.
  - When there are multiple consumers for a producer and vice-versa.
  - When historical data is required.

Data Buffering



SC - Sensor Controllers

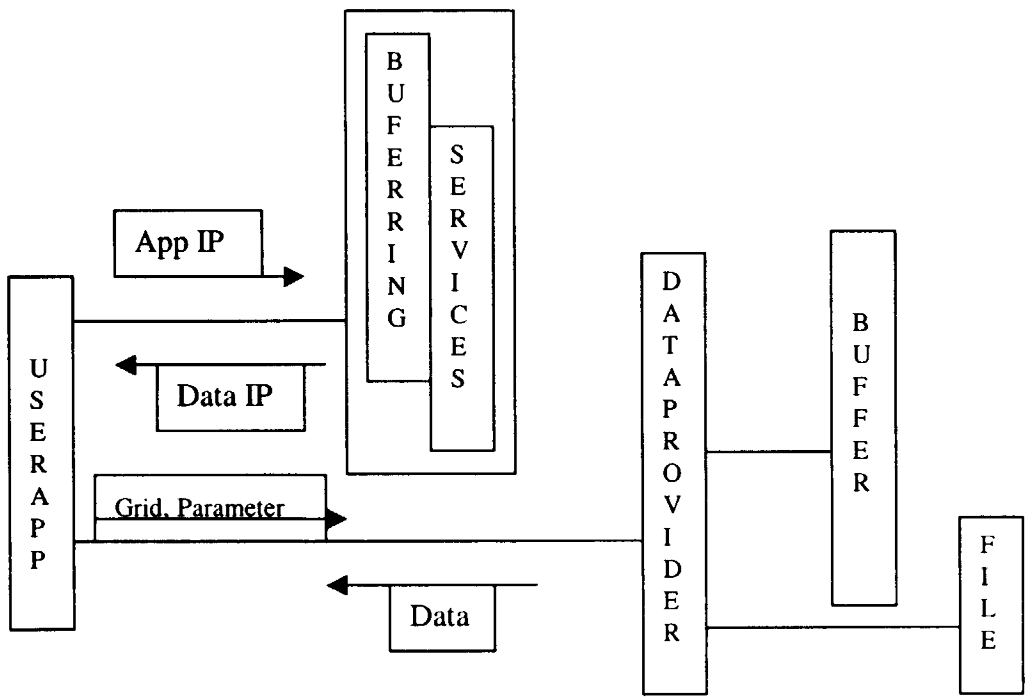
RDS - Buffers

RF - Remote File

S - Sensors

Figure

Data Retrieval



Figure

Performance Enhancement techniques in the HUMS Reference Model

- Efficient design and fabrication.
- Duplication or Replication of data and processes.
- Hand-shaking between communicating components (e.g., are you alive messages).
  - *Each of the above mechanisms has some advantages and disadvantages.*

Efficient Design and Fabrication

- Advantages
  - *The best method to develop fault-tolerant and reliable systems.*
- Disadvantages
  - *Significant design effort.*
  - *Cost is prohibitive.*
  - *COTS products usually do not conform to the high reliability requirements of avionics.*

Replication of Components

- Replication is achieved through computation replication and (or) data replication.
- A version of replication called N-version programming is used in software.
- Advantages
  - *Provides reliability over unreliable systems.*
- Disadvantages
  - *They involve more computation and communication.*
  - *Not viable in systems that have stringent space and weight constraints.*

Hand-shaking between Communicating Components

- A reliability enhancement technique to ensure reliable data transfer between data producers and consumers.
- Disadvantage
  - *They require producers and consumers to be pro-active (i.e., they should be able to identify loss of messages and respond to queries).*
  - *They result in higher network traffic requiring higher network bandwidth.*
  - *They are not feasible for highly periodic real-time data transfer.*

Performance Enhancement at the Sensor Layer

- Reliability enhancement mechanisms that can be used, depend on *sensor capabilities* and *protocol issues*.
- Case 1: Sensors are simple and communication is simplex.
  - Sensors are not concerned about the data reaching the appropriate destination.
  - They are not equipped with capabilities to respond to inquiries or acknowledgements.
  - Do not provide control information or error control mechanisms.
- *The reliability enhancement mechanisms possible when this is the case are*
  - *Better fabrication of sensors. This increases sensor cost.*
  - *Replication of sensors. This results in a large number of sensors.*

Perf. Enhancement, Sensor Layer, contd.

- Case 2: Sensors are complex and communication is duplex
  - Digital logic is embedded in the sensors for error control
  - Communication is duplex and sensors have the capability to respond to requests from higher layers.
- *The reliability enhancement mechanisms possible when this is the case are*
  - *Better design of sensors. Sensors are a generation more complex.They should also have buffers to temporarily store data for retransmission requests. Increases sensor cost and weight.*
  - *Sensors provide control information and respond to ACKS/NACKS from higher layers. This overhead consumes bandwidth and effectively reduces the capacity available for the payload*
- Issues and solutions in both cases involve a balance to be achieved among cost, performance and space/weight.

Distributed Robust Systems for Structural Health Usage and Monitoring

Ravi Mukkamala

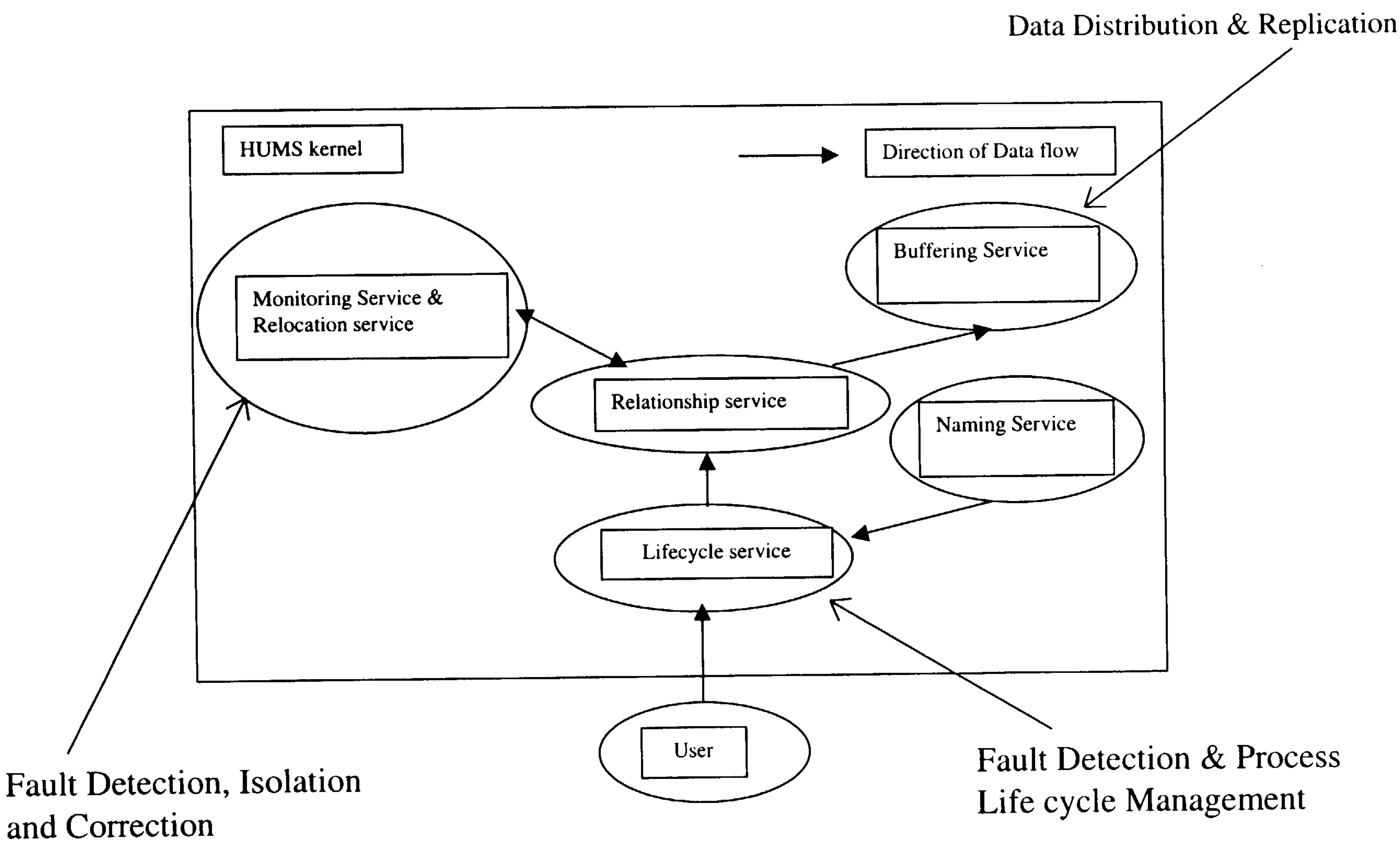
Kailash Bhoopalam

Old Dominion University, Norfolk VA

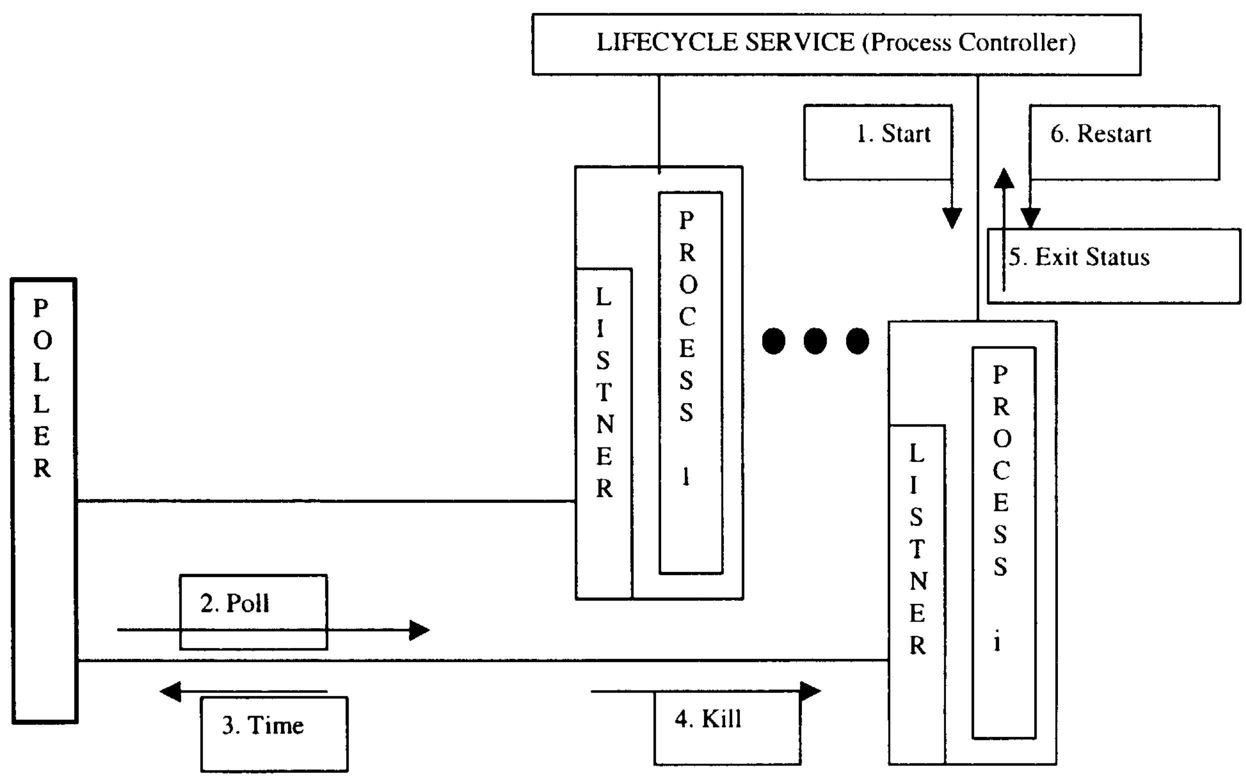
Building Reliable Systems

- Structural Health & Usage Monitoring Systems?
- Component Reliability
  - Identify/Isolate h/w failures
    - Distributed Architecture
- Communication Reliability
  - Identify/Reduce transmission losses
    - Reliable Protocol
- Process & Data Reliability
  - Identify/Isolate/Rectify s/w & signal failures
    - HUMS Kernel

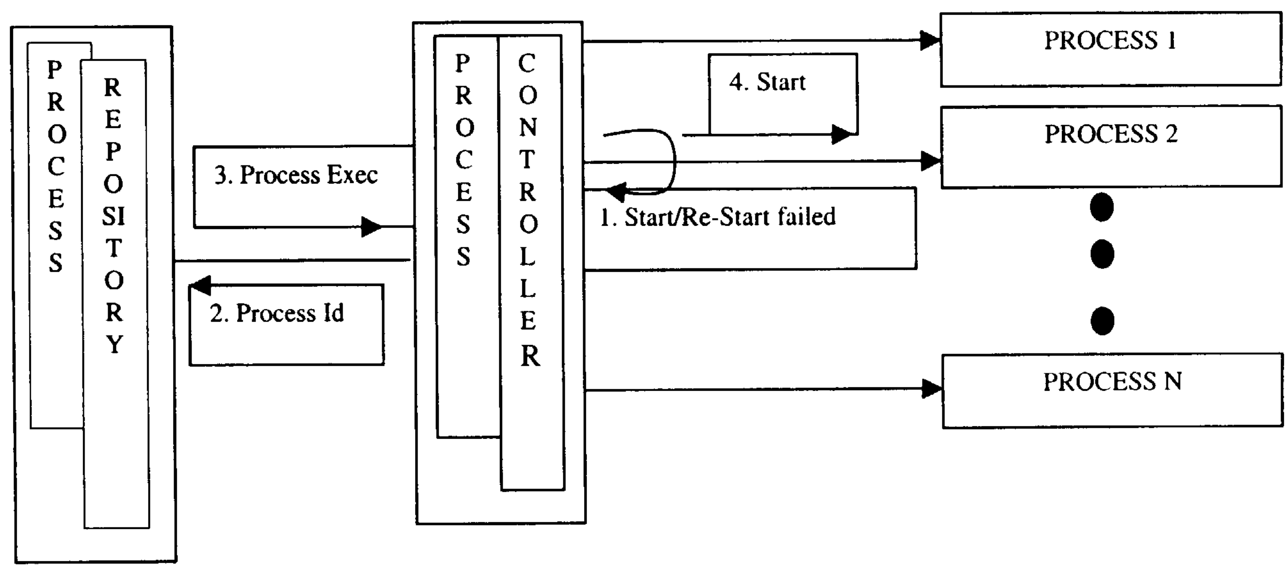
The HUMS Kernel



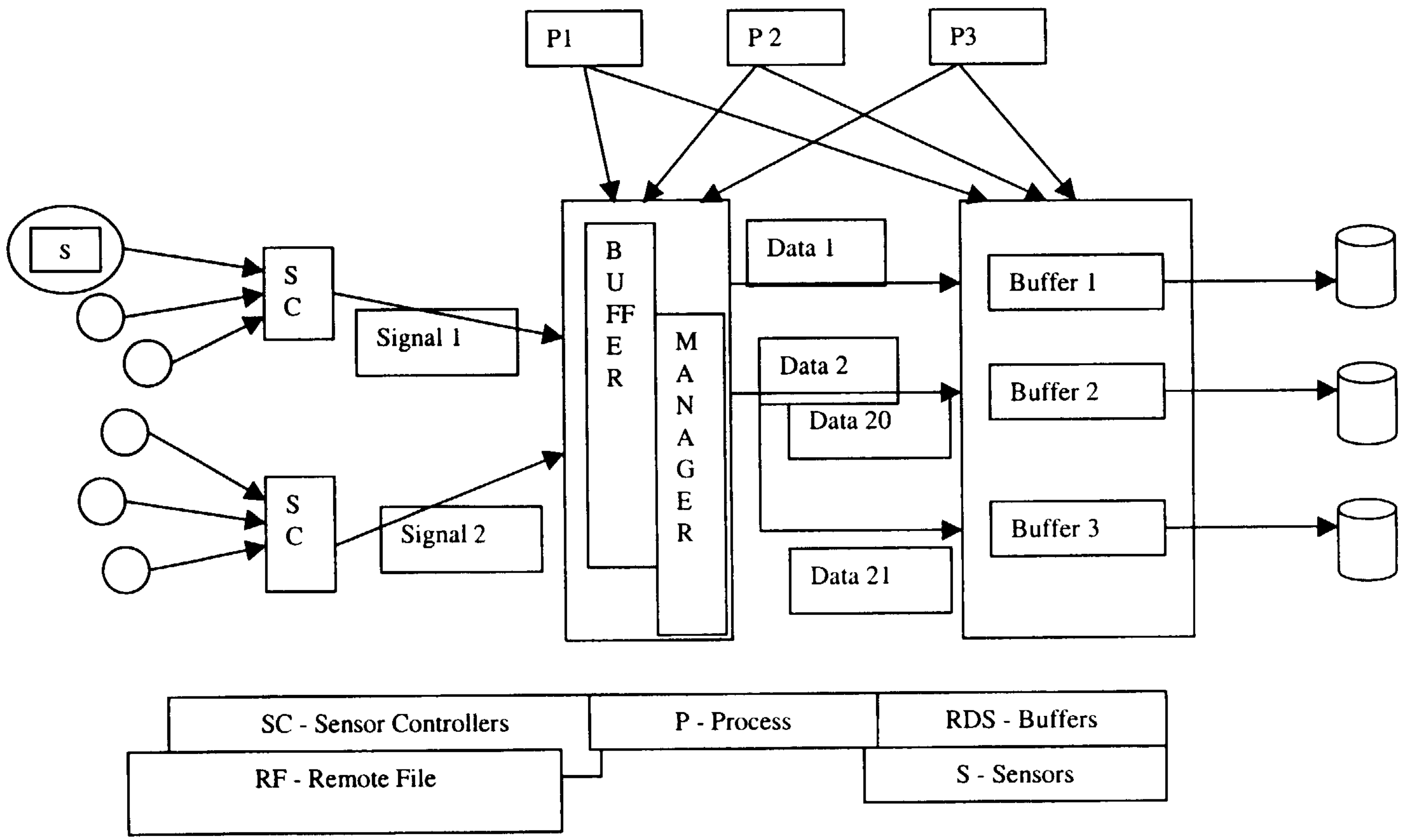
Monitoring & Relocation Services



Monitoring & Relocation Services



Buffering Services



Design and Analysis of a Scalable Kernel for Health Management of Aerospace Structures

Dr. Ravi Mukkamala,

Kailash P. Bhoopalam, Srihari Dammalapatti

Department of Computer Science

Old Dominion University

Norfolk, Virginia

Mukka, kbhoopal, damma_s@cs.odu.edu

Work supported by a grant from NASA LaRC

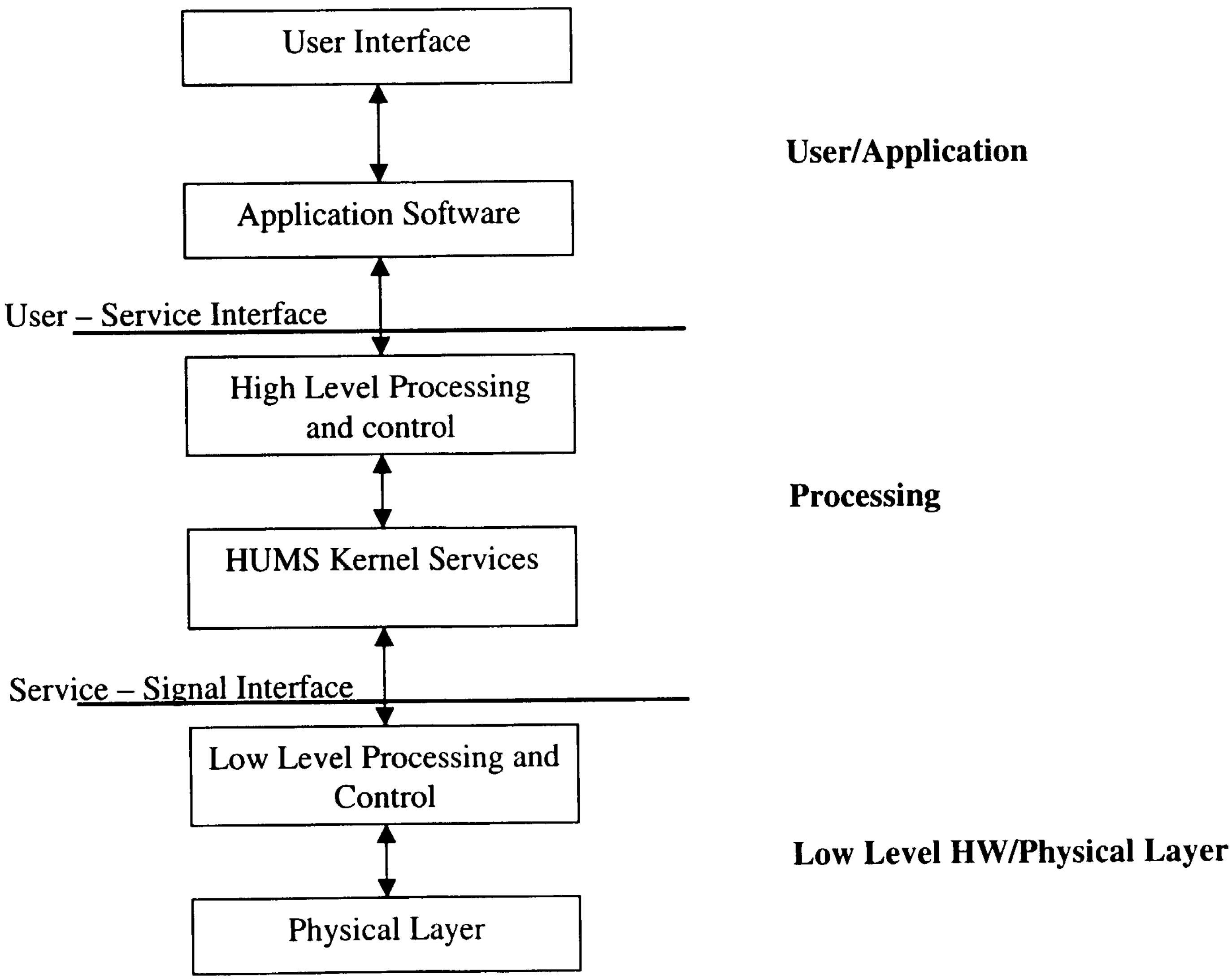
Introduction

- Introduction
- The HUMS Reference Model
- The HUMS Kernel
- Reliability & Scalability Issues in the HUMS kernel
- Architectural Proposal

Reference Model

- Motivation
  - Provide a template for Architecture description.
  - Communicate System Requirements for invitation of Architecture Proposals.
  - Provide a Baseline Evaluation criteria.
  - Encourage a common Architecture Description Language.

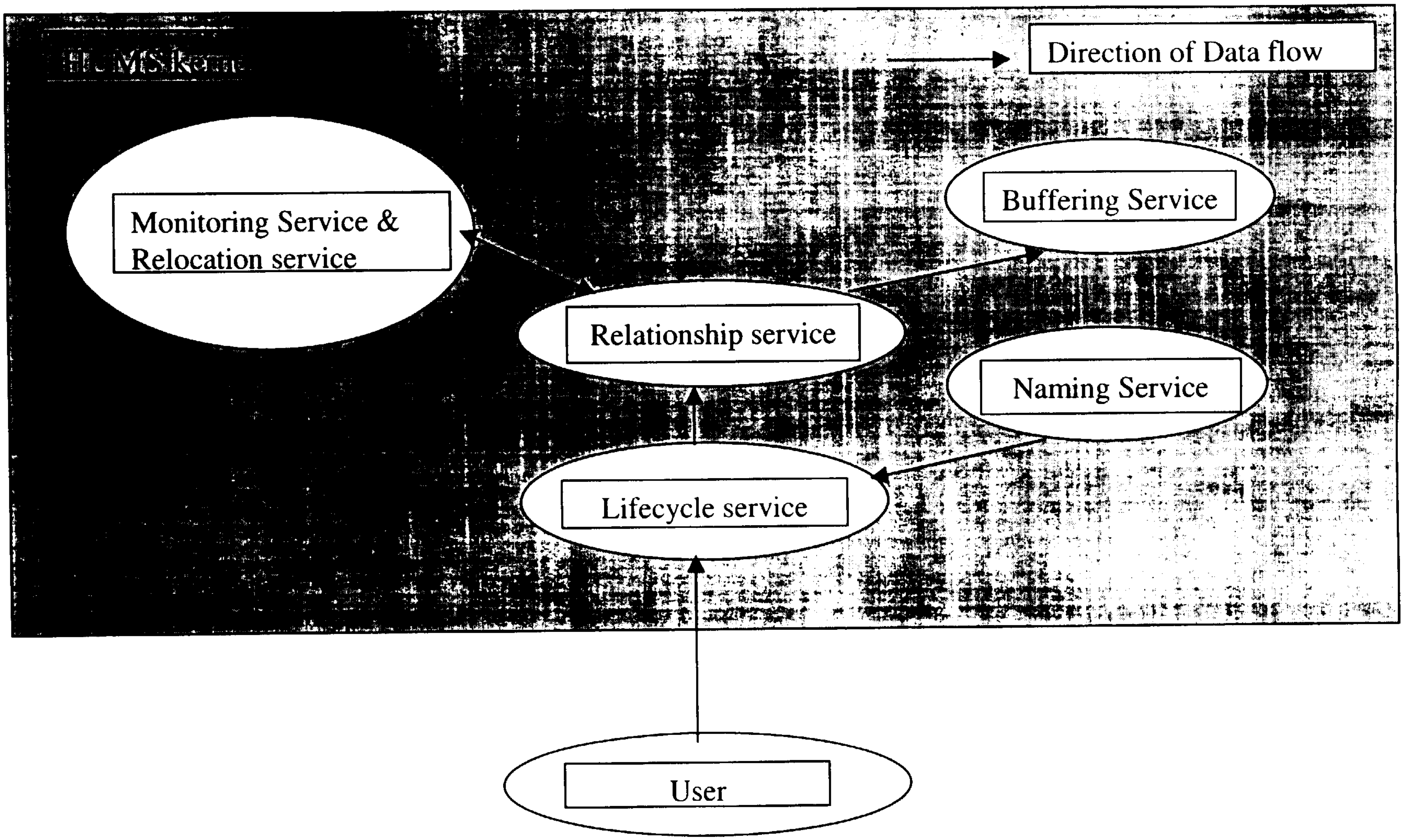
Reference Model



The HUMS Kernel

- The objectives of the HUMS kernel are provided as services to the HUMS systems
  - Reliability (Reduce process failures)
  - Robustness (Allow processes to react gracefully during component failure)
  - Provide Services for system maintenance (addition and deletion of components)
  - Provide services for access to sensor data

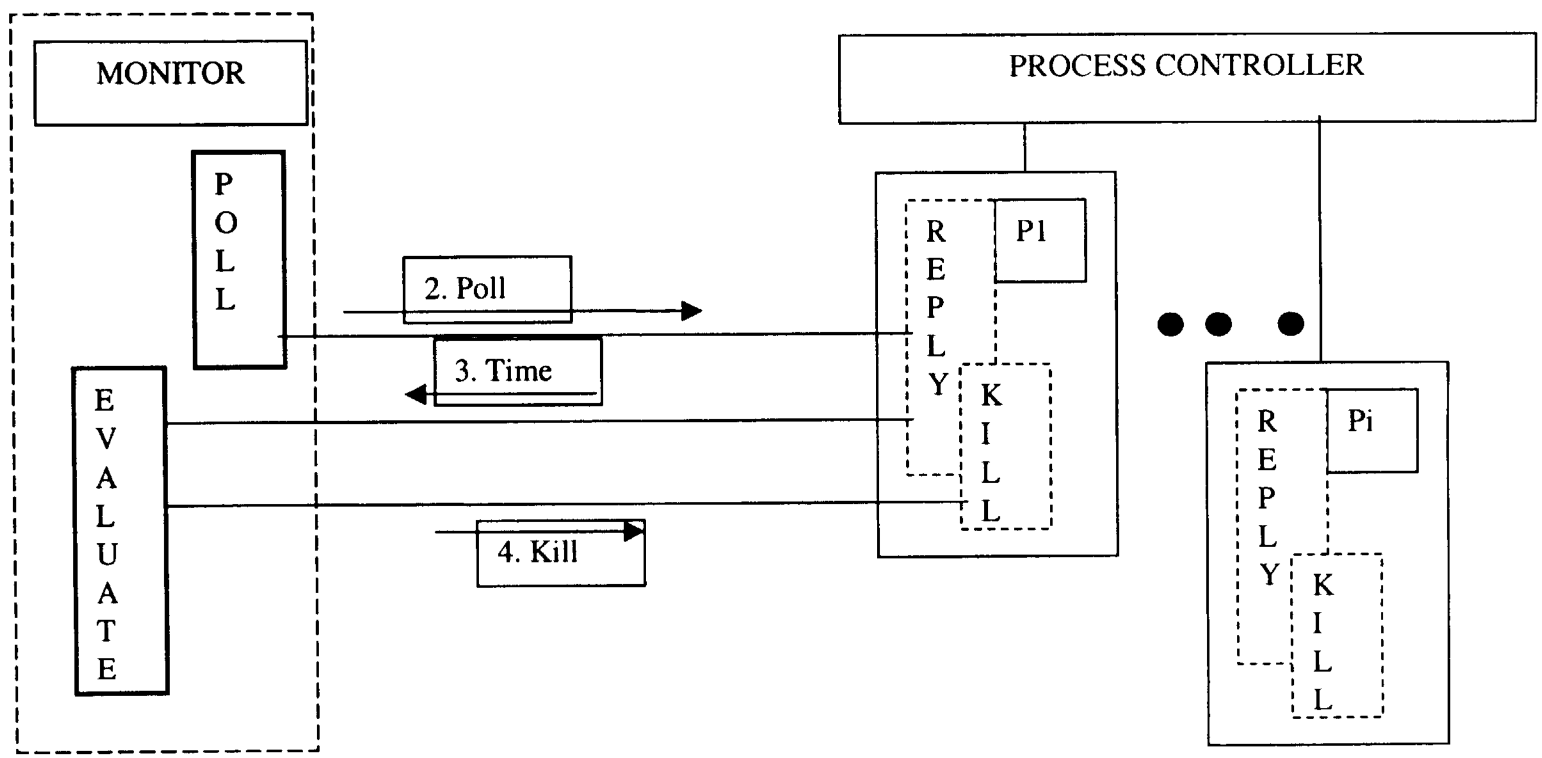
The HUMS Kernel, Interactions



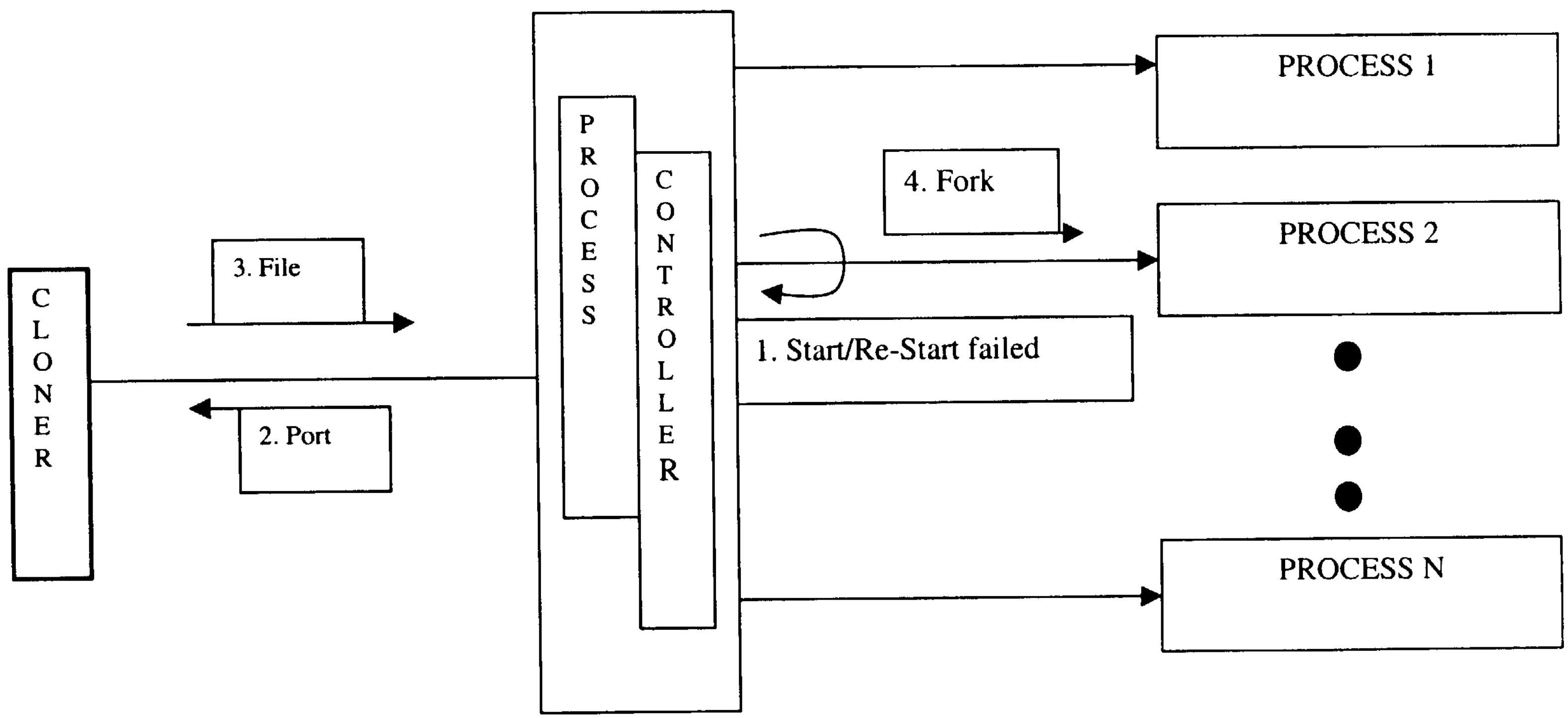
Monitoring and Relocation Services

- Purpose
  - Increase the reliability of the services in the system.
- The Monitoring and Relocation services consist of the following functionalities
  - Polling & Evaluation
  - Cloning & Relocation

Polling and Evaluation



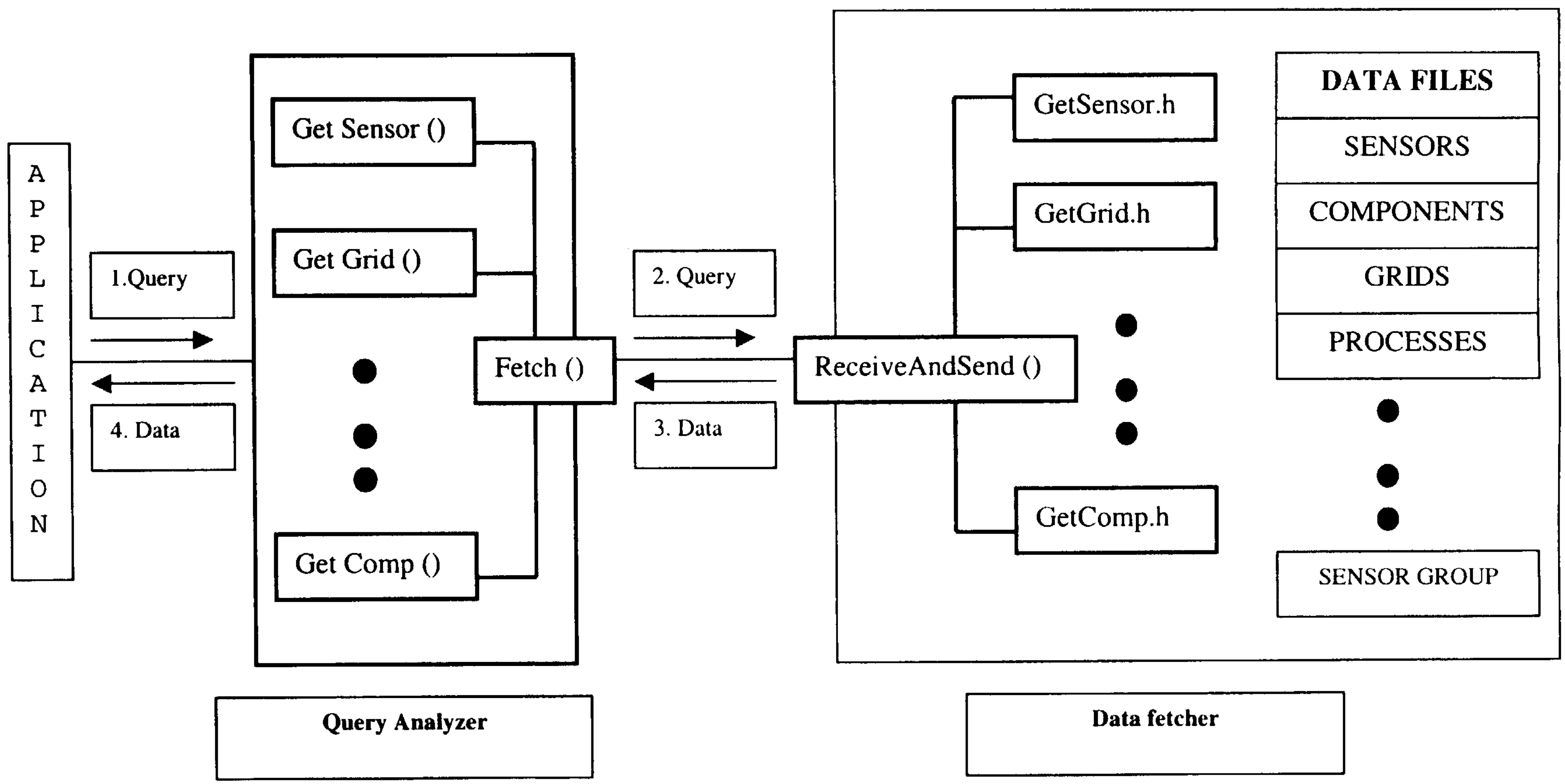
Cloning & Relocation



Relationship Service

- Maintains a hierarchy of Structural components of an aircraft.
- Maintains a relationship of HUMS components, the relationships can be broadly classified into the following
  - <Component, Grid>
  - <Grid, Sensor>
  - <Processor, Process>
- Mechanisms for storing and retrieving data.

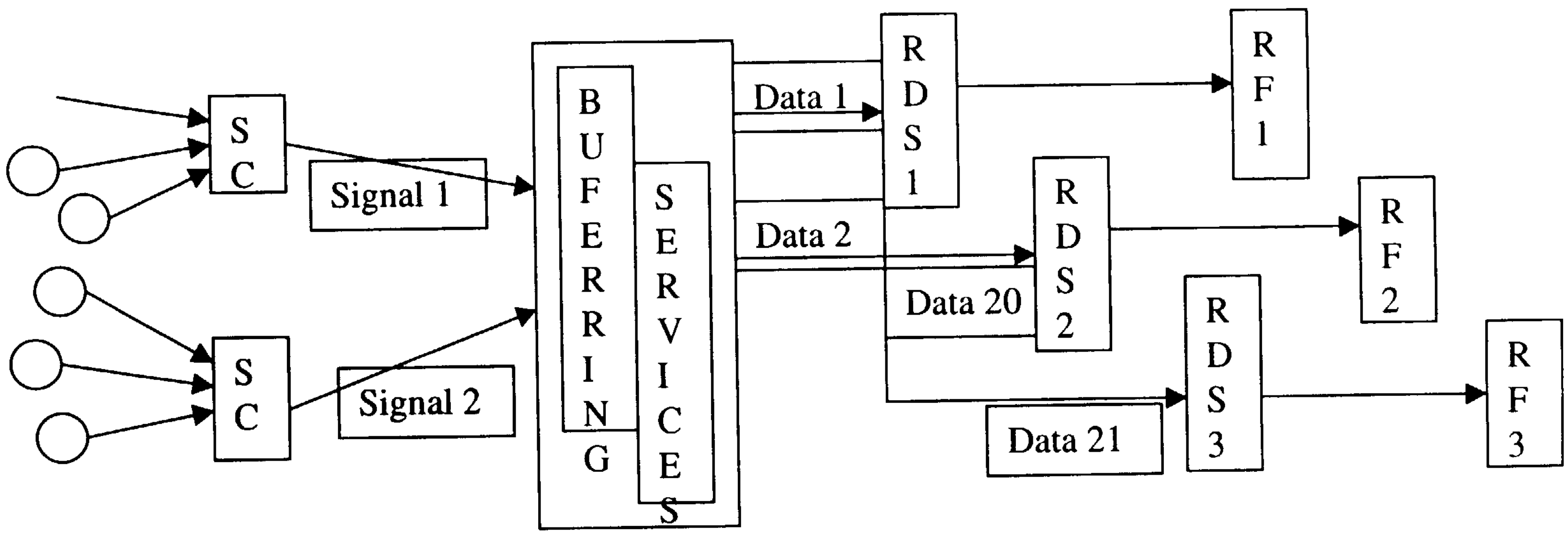
Relationship Access Mechanism



Buffering Services

- The buffering services provide for sensor data to be stored so that they can be accessed under the following conditions:
  - When a Process's requirement for data is not in synch with the emission of data by the sensors.
  - When there are multiple consumers for a producer and vice-versa.
  - When historical data is required.

Data Buffering



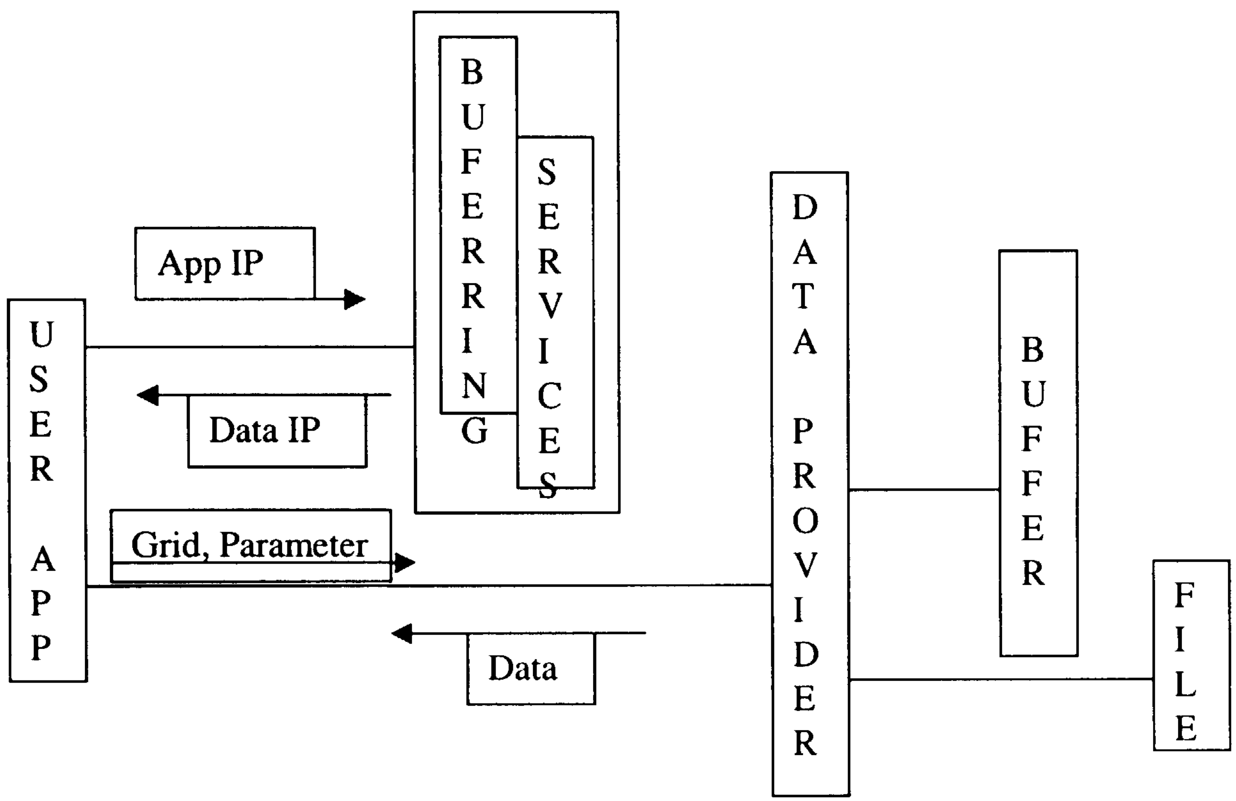
SC - Sensor Controllers

RDS - Buffers

RF - Remote File

S - Sensors

Data Retrieval



Performance Enhancement techniques in the HUMS Reference Model

- Efficient design and fabrication.
- Duplication or Replication of data and processes.
- Hand-shaking between communicating components (e.g., are you alive messages).
  - *Each of the above mechanisms has some advantages and disadvantages.*

Performance Enhancement - Kernel Layer

- Lifecycle Service
  - Reliability Issues
    - The process controller is *replicated* in processors that have processes whose lifecycle have to be monitored.
    - May consist of the cloner and the redirection service. These services can be *replicated* and *monitored* for reliability enhancement.
  - Flexibility issues
    - Simple (Bootstrapping and process restarting).
    - Complex (Processor utilization, cloning, redirection)

Perf. Enhancement: Kernel Layer, contd.

- Monitoring and Relocation service
  - Reliability issues
    - When the number of critical components in the system increases, the load on the monitoring service also increases.
    - The monitoring service can be *replicated* at different processors. This provides the following advantages:
      - Enhances the reliability of the monitoring service.
      - Distributes the monitoring load.
  - Flexibility issues
    - Simple (Log process failures).
    - Complex (Monitoring, Adoption, Relocation).

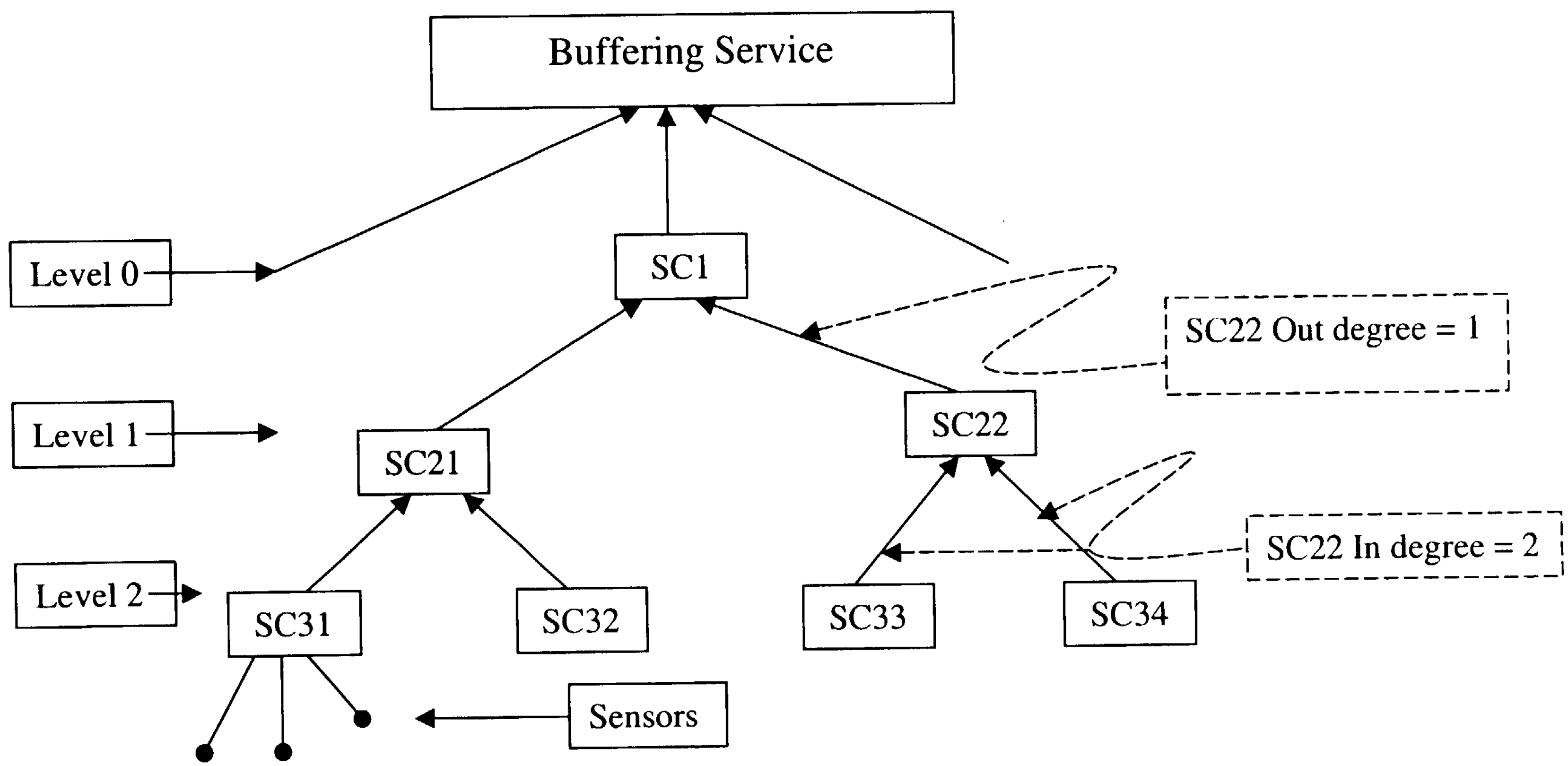
Perf. Enhancement: Kernel Layer, contd.

- Relationship service
  - Reliability and Access time Issues
    - *Replicating* the data in multiple locations can enhance reliability. In addition, relationship service can be *monitored* for increased reliability.
    - *Replicating* relationships of interest nearer to where they are frequently accessed. This would increase the response time of the relationship service.

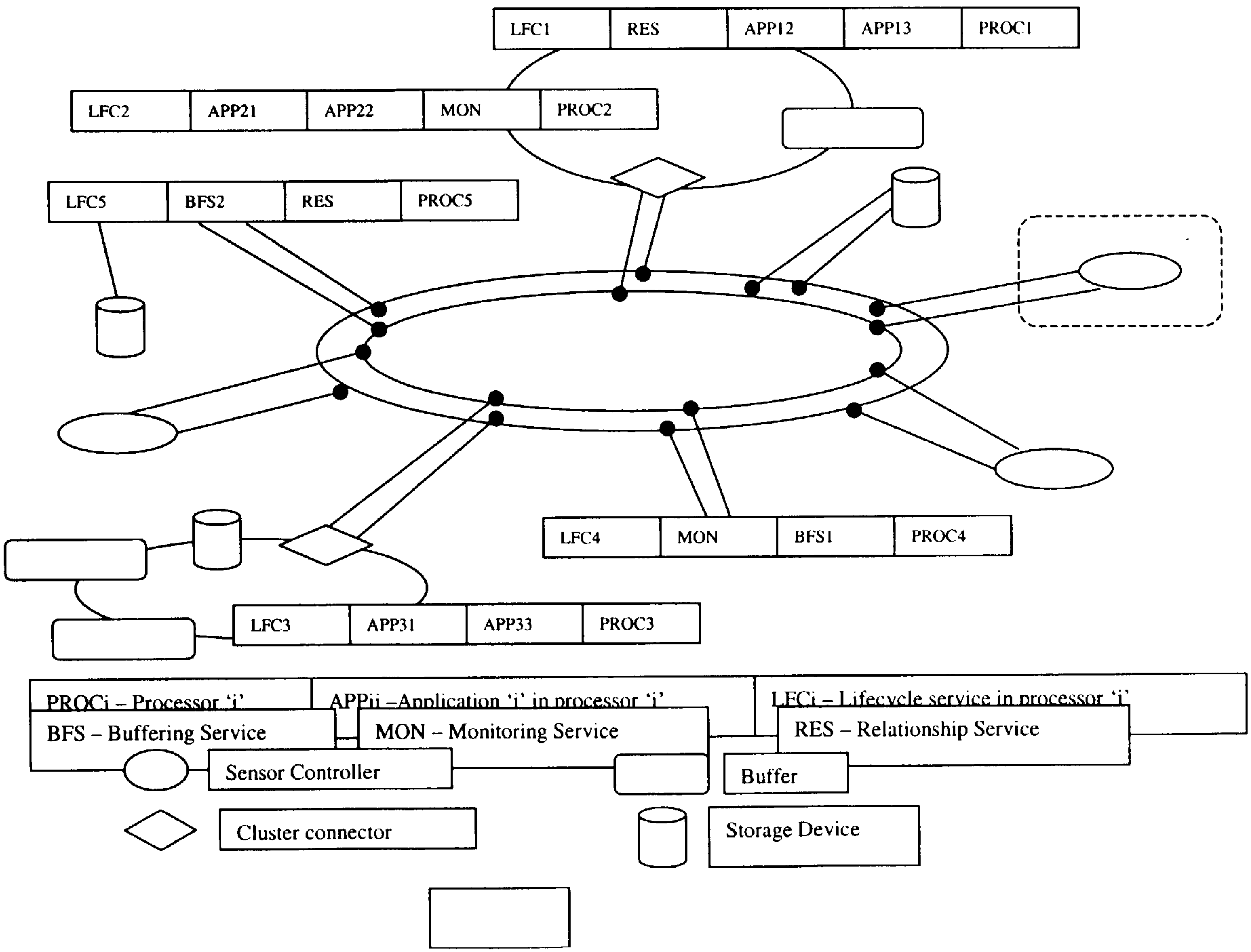
Perf. Enhancement: Kernel Layer, contd.

- Buffering service (Service and the Buffers)
  - Reliability and Data Access time Issues
    - The Buffering service can be n-*replicated* and the leader (active service) can be decided by vote.
    - Different buffering services can can cater to different sets of sensor controllers.
    - The buffers can be *distributed* in the system and data from can be *replicated* in more than one buffer.
    - Distribution of data decreases data retrieval time, since search space for data retrieval is less

A General Performance Scenario



Architectural Proposal



Architectural Proposal, contd.

- Cluster Organization: A cluster might consist of the following components
  - A set of processors with frequently interacting processes.
  - A cluster would also have buffers that contain data that is frequently accessed by the processes within the cluster.
  - Part of the relationship service of interest to the processes can be replicated in the cluster.
  - If the processes in a critical to the system the monitoring services can be replicated in that clusters.
  - Storage devices that store data analyzed by the processes.

Architectural Proposal, contd.

- Advantages of clustering mechanism
  - It ensures that data from the sensors are stored with high degree of reliability (because of data distribution and replication) in the buffers.
  - Provision of clusters allows heterogeneous sub-systems to be connected to the FDDI backplane.
  - Increases system performance because of data locality (data required by the applications in the cluster are available in buffers located within the cluster)
  - Distributes monitoring load and allows monitoring heterogeneity.
  - Failure of clusters result in partial failures. Failure of a cluster does not affect other clusters.

Future Work

- Formal specification methods in HUMS
- Exhaustive, HUMS Evaluation methodology using
  - Scenarios
  - Check lists
- Assets distribution among stakeholders in HUMS.

Distributed Scalable Architectures for Health Monitoring of Aerospace Structures

Ravi Mukkamala

Department of Computer Science

Old Dominion University

Norfolk, Virginia

mukka@cs.odu.edu

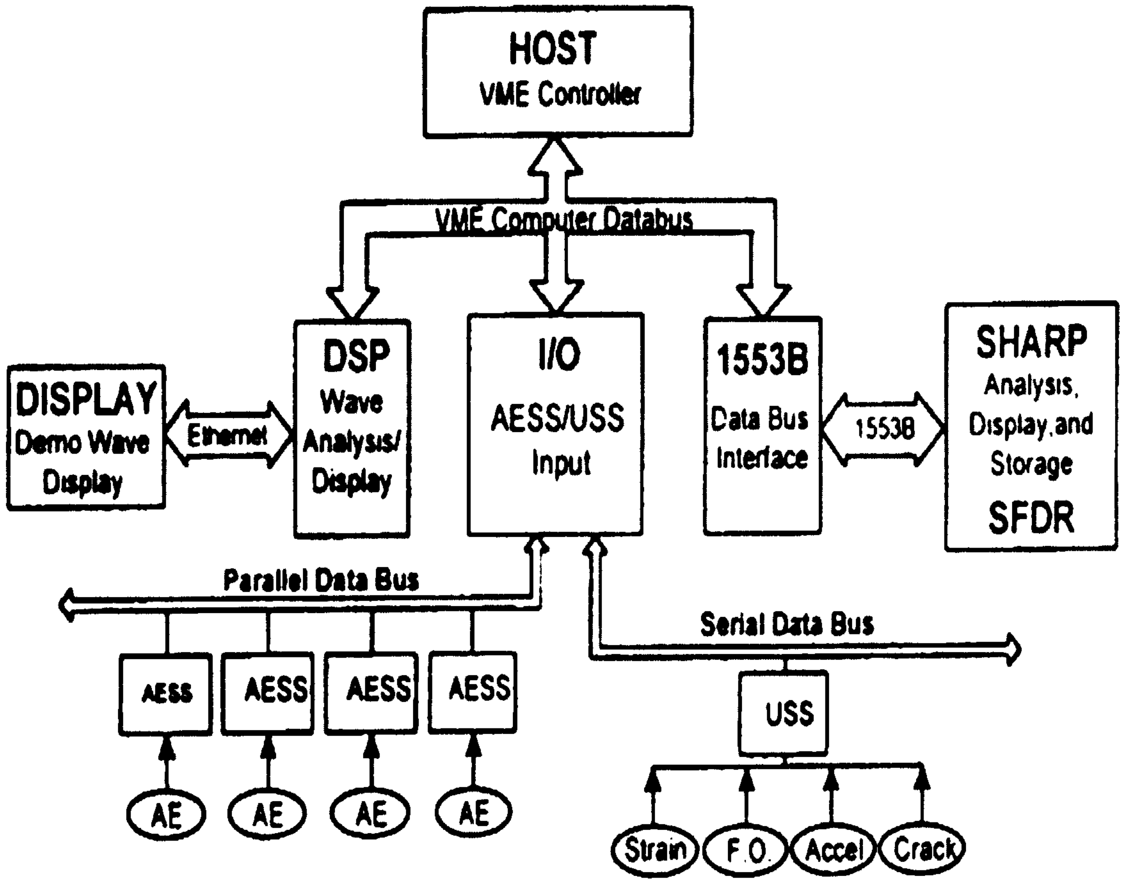
Work supported by a grant from NASA LaRC

ORGANIZATION

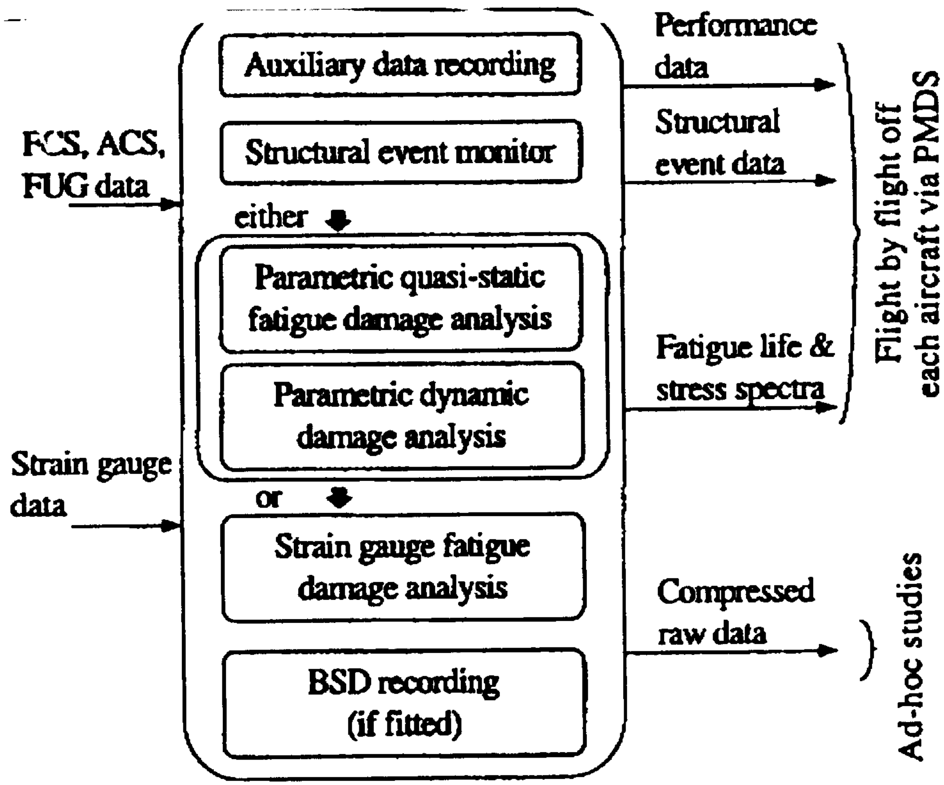
- Introduction
- Background
- Characteristics of HUMS
- Tools for Achieving the Characteristics
- Proposed Reference Architecture
- Evaluation Methods for HUMS Architectures
- Future Work

Example SHM Systems

SHM System developed as part of ASIP

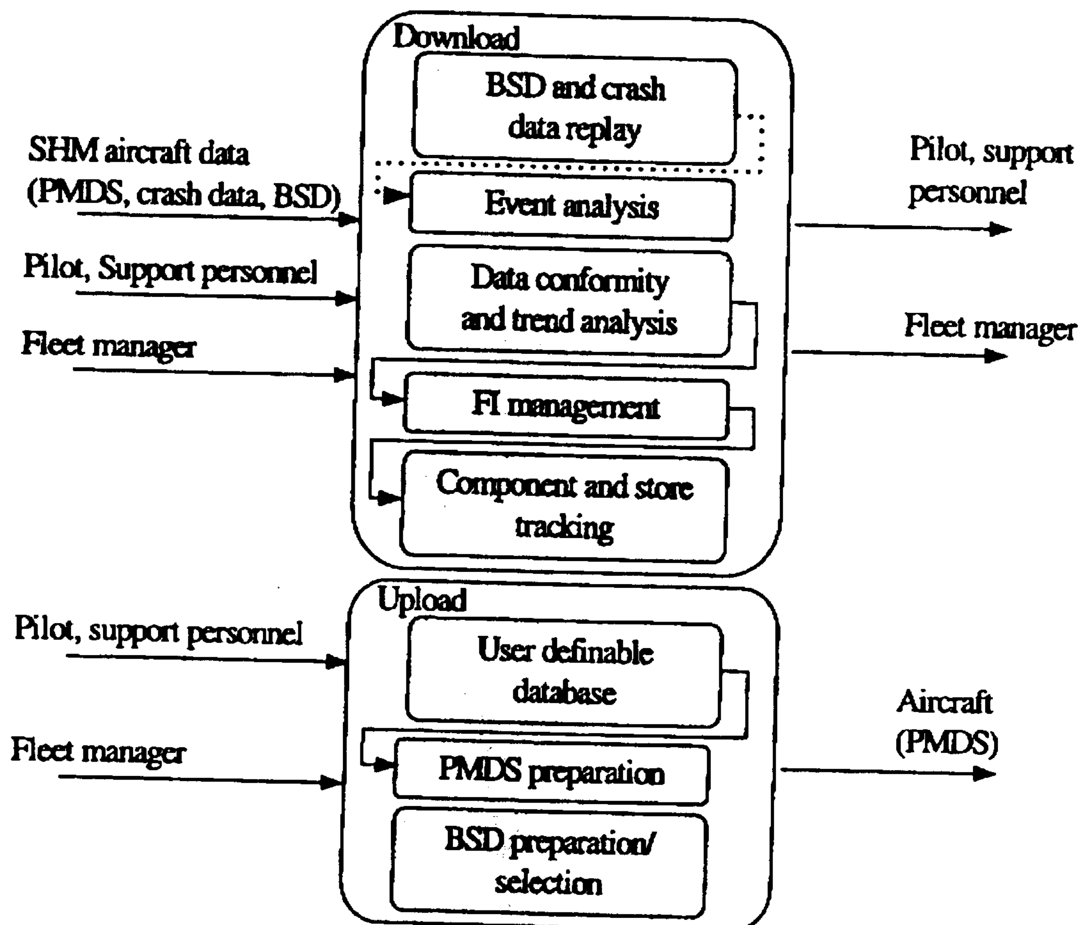


EF 2000 SHUM



. Schematic diagram of aircraft SHM system

EF2000 SHUM (Ground System)



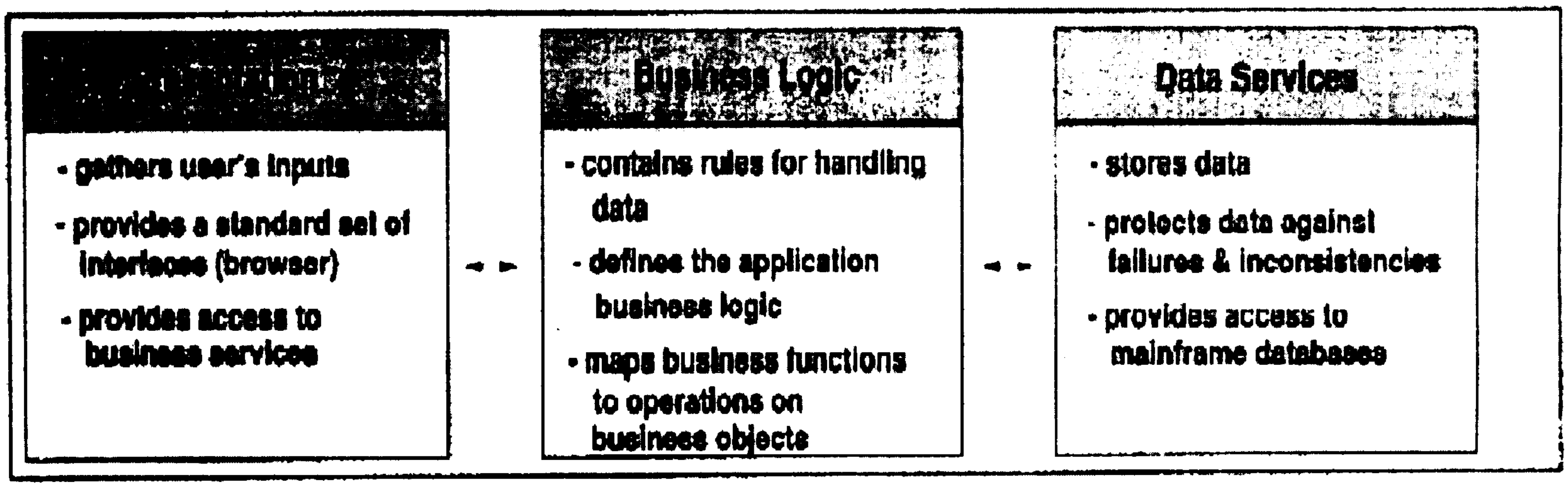
Schematic diagram of ground SHM system

Characteristics of HUMS

- Scalability
- Openness
- Flexibility
- Robustness
- Extendibility
- Incorporating legacy systems
- Intelligent monitoring

Multi-tier Architectures

- Multi-tier architectures are useful in obtaining scalability, availability, security, and integration with legacy systems. The following diagram (Menace and Almeida 2000) describes a typical three-tier architecture.



Typical Multi-Tier System Architecture

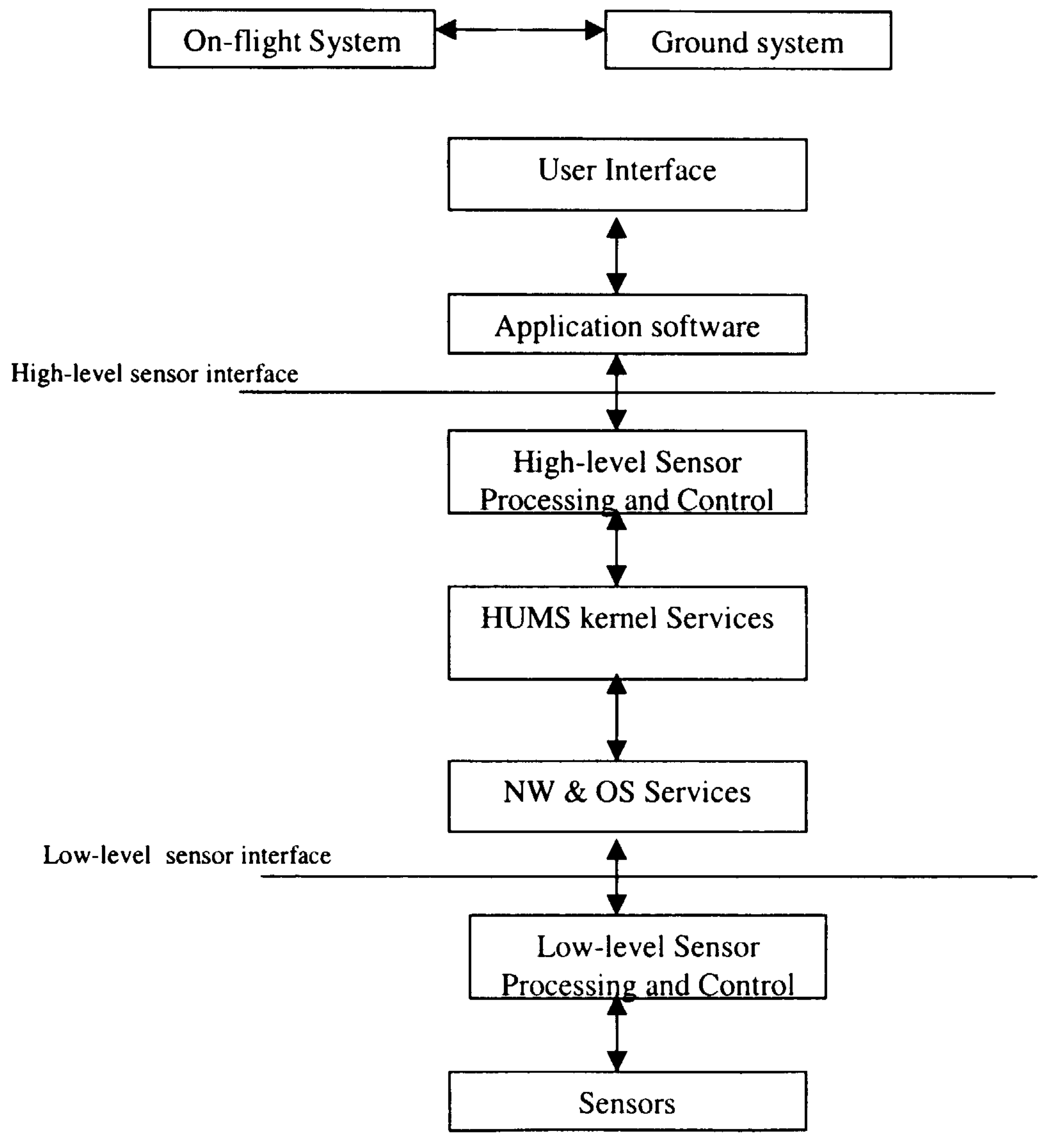
Use of Clusters for Scalability and Robustness

- A group of nodes/processors and resources make up a cluster.
- Members within a cluster mainly communicate via message passing.
- A cluster offers an interface that includes services (e.g., storing/retrieving data, data processing/analysis).
- Clusters have the advantage of scalability and fault-tolerance at the same time being cost effective.
- The idea of cluster computing is being used extensively in the commercial world in applications such as E-commerce and in scientific computing applications with large computational problems.
- Scalability, transparency, and reliability (or availability) are the main goals of a cluster software
- The concept should be equally appealing to HUMS architects and designers

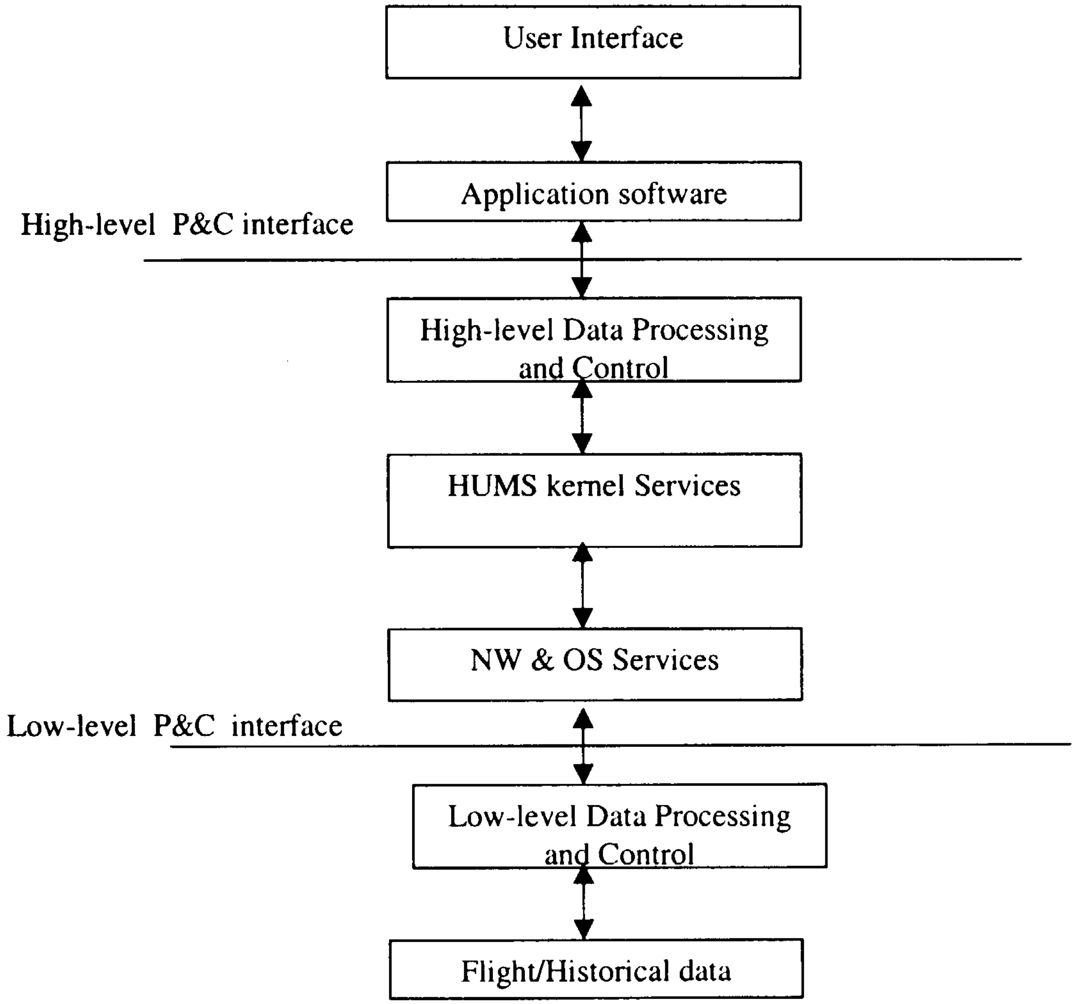
More Tools for Scalability

- Cloning: The same software is run on several processors/nodes.
- The set of clones supporting a certain service are referred to as RACS (Reliable array of cloned services).
- A clone may be implemented in several ways (i) the data is shared but the processing is independent or (ii) no data sharing through data replication
- Partitioning: the data and/or functionality of a system is divided among several nodes
- The set of partitions supporting a certain service are referred to as RAPS (Reliable aray of partitioned services)

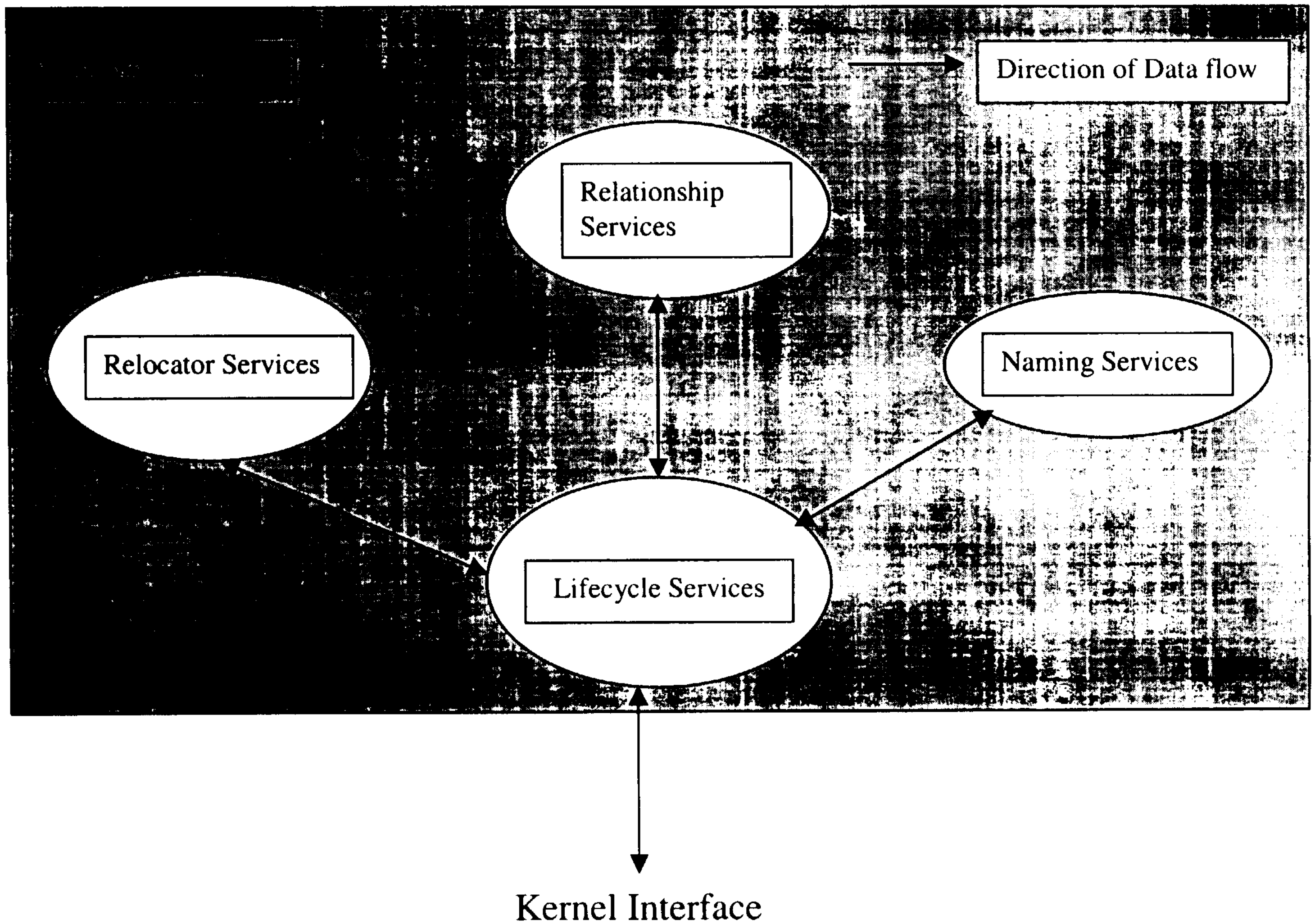
Proposed Distributed HUMS Reference Model
On-flight System



Proposed Distributed HUMS Reference Model
Ground System



HUMS Kernel Modules



HUMS Kernel--Services

- Provides system services specific to HUMS, and not generally provided by a general OS
- It provides distributed system functionality needed by the HUMS architectures
- In our preliminary, we identified four useful kernel services:

  * Naming Service
  * Lifecycle Service
  * Relationship Service
  * Relocator Service

Evaluation of HUMS Architectures: What is Required?

1. A model of the architecture
2. A schematic illustration of the interaction between components or blocks in the architecture, similar to an event-diagram
3. Processing and communication load induced by the components and blocks on the system
4. Expected resources to be available in the underlying system
5. What are the questions to be answered by the architecture? Typical questions may be:
   - The number of sensors that the architecture can support (i.e., identifying sustained peak loads and the bottleneck components.)
   - Response time and/or throughput of services/system

Evaluation of HUMS Architectures: Who is Involved?

- **Intended users:** To describe anticipated patterns of use of the system
- **System designers:** To identify the interaction between the components in terms of which module invokes which modules and how frequently
- **System implementers:** To specify the resource requirements for each module in system
- **Configuration planners:** To translate the system-independent resource requirements into configuration-dependent terms
- **Performance analysts:** To synthesize the information and construct an analytical model to answer the questions.

Evaluation of HUMS Architectures: The Steps

- **Modeling cycle:** Validation, projection, and verification
- **Understanding the objectives**
- **Workload characterization**
- **Sensitivity analysis:** Robustness of the results to the assumptions in question

Scalability Analysis

- Synonymous with bottleneck analysis.
- One of the primary reasons a system cannot scale up is that while there is one bottleneck component in the system, we are attempting to scale up another non-bottleneck component.
- In that sense, while the cost of the scale up has increased results have not improved to the same extent.
- It could also be related to the nature of the application that dictates the ceiling on the performance (e.g., throughput) of a system. For example, when an application is **highly sequential** in nature, adding extra processors is not going to significantly improve its response time.

More on Scalability

- When we say that a piece of software must be scalable, we mean that it must be capable of being deployed efficiently at both large and small scales.
- Alternately, an architecture is scalable when it provides adequate service levels even when the workload increases above expected levels (Scaling up and Scaling out)
- In addition, over time, we must expect that it should be possible **to add capacity** *to* either **support more users**, *or* **enhance the quality of service**, *or* **both**.
- If added resources do not increase the scale of operation (number of users, quality of service, or both) in proportion to their cost, then the system will become uneconomical (or infeasible) to operate. This is the problem of scalability.

Limiting Factors of Scalability

- While multiprocessors are a tool to scaling a system, overhead of distributed/parallel processing limits its utility.
- **Sources of overhead:**
  * Time to communicate and/or synchronize
  * Load imbalance
  * Additional computations
- **Possible culprits:**
  * Inefficient underlying system support (BW, MIPS, etc.)
  * Inherent sequential nature of a problem
  * Poor algorithm design
  * Architectural limitations

Measures of Scalability

- **P-scalability** = $[Power_2/Cost_2]/[Power_1/Cost_1]$
  $= [\lambda_2/(T_2C_2)]/[\lambda_1/(T_1C_1)]$

- **G-scalability** = $F(QoS_2, C_2)/F(QoS_1, C_1)$

- **Isospeed Scalability** = $T(W_1,P_1)/T(W_2,P_2)$ where W is useful work done, P is the resource size such as the number of processors, and T is a performance metric such as response time or time for completion.
- **Average-speed** = $W/(W+T_0(W,P))$ where T_0 is the overhead due to distribution or parallelism

Scenario-based Evaluation of HUMS Architectures

- Scenarios illustrate the types of activities the system must support and the types of changes designers and users expect the system to undergo over time.
- Direct or indirect scenarios
- Enables better understanding of the different needs of the stakeholders and to recognize in the early stages of design whether or not the proposed architecture meets them.

Scenario-based Evaluation---Examples

- Addition of new sensors to existing components
- Inclusion of new components for sensing.
- Handling sensor failures (fail-stop)
- Handling misbehaved sensors (e.g., spurious generation of data)
- Replacing existing sensors with different ones
- Failure of processors
- Both synchronous and asynchronous sensor outputs
- Lost messages
- Etc.

Robustness

- Def: The measure or extent of the ability of a system to continue to function despite the existence of faults in subsystems or components.
- Other related terms are: Reliability, Accessibility, Accuracy, Correctness, Availability
- In multi-tier (or hierarchical) architectures, we should evaluate the robustness of each layer. Certainly, the robustness offered by lower layers, affects the robustness of higher layers. The robustness at the user-interface level is what is the "system robustness"

Sample Robustness Measures in HUMS

- RO1: Probability with which a component's (e.g., left wing, engine) status is recorded in the storage (applicable at lower-layers)
- RO2: Probability with which a higher-layer algorithm is guaranteed to receive sensor inputs within a given time interval (Tests timeliness)
- RO3: Probability with which the status of at least k out of m required components is available (applicable at middle and higher layers)
- Other metrics (at component, subsystem, service, or system level): Rate of failures, Mean-time-between-failures, Mean-time-to-recover

Extendibility

- **Def:** The ease with which a system or component can be modified to increase its capacity or functionality.
- Functional extendibility
- Network extendibility
- Processor extendibility
- Storage extendibility
- Sensor extendibility

Flexibility

- Often, definitions of flexibility and extendibility overlap
- IEEE Def.: **Flexibility is the ease with which a system or a component can be modified for use in applications or environments other than those it was designed for.**
- Example scenarios for HUMS:
  - Can a system designed to handle periodic sensor signals be modified to handle aperiodic sensor signals?
  - Can a system currently monitoring a fixed set of variables, be modified to handle additional variables?
  - Can a system with functions that are currently performed on ground be modified so they can be performed on-flight?
  - Can a system that currently requires its sensors to asynchronously send signals to a processor be modified so that the processor could now selectively poll them for input?

Future Work

- Identify paradigms that are useful building blocks for HUMS systems.
- Develop a framework by which a HUMS architecture can be described to enable its evaluation.
- Define quantitative and qualitative metrics of HUMS architecture evaluation.
- Develop methods to evaluate the metrics given an architectural description.
- Use the methods to evaluate some of the existing architectures and identify the bottlenecks.
- Develop a suite of model architectures using the above techniques.